CHAPTER 6

Securing the Service

Ah, security. You knew we'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 end up on the front page, or completely go under, when security is implemented poorly. You simply can't afford to mess it up!

Fortunately, because we are dealing with a RESTful service that is anchored on HTTP, we can leverage widely-used security mechanisms (some of which have been in place for years) for the more complicated and risky parts of the security architecture. In this chapter, we will highlight some of those mechanisms as we add security to our task-management service. Along the way, we'll also highlight some useful design approaches and ASP.NET Web API features that, though not intrinsically tied to security, seem to fit well in the context of this subject.

The Main Idea

In Chapter 5, we implemented a scenario where the user created a task. In this chapter, we are going to return to that scenario and add security to it. We are also going to implement a few more scenarios so that we can more fully illustrate the design and implementation of security in the context of a service built using ASP.NET Web API. Table 6-1 summarizes the service scenarios we will cover in this chapter, including the authorization level required of the user.

Table 6-1. *Scenarios Used to Illustrate Security*

Scenario	Required User Role
Create a task	Manager
Activate, complete, or reactivate a task	Senior Worker
Get a task	Junior Worker

Before we go any further, though, let's agree upon some basic terminology:

- **User** The end user of the task-management service. This may or may not be an actual human being. We will refer to the user as "he" just for simplicity.
- **Caller** The application that invokes the task-management service on behalf of the user (e.g., a browser). Fiddler was the caller in Chapter 5.

Now that we've agreed upon that terminology, let's get things started by breaking the security of the service into two parts: authentication and authorization. *Authentication* answers the question, "Is the user of the API service who he claims to be?" And *authorization* answers the question, "Is the user allowed to do what he is trying to do?" In other words, authentication establishes the user's identity, and authorization enforces the user's permissions.

Authentication

The first thing the service must do when it receives a new web request is verify the user's claim of identity. We do this by validating the credentials supplied by the user. Via these credentials, the caller provides two basic pieces of information: who the user claims to be, and how that claim can be verified. You likely do this every day when you log into your computer. You claim to be [your name here], and the password you enter on the login screen provides proof/validation of that claim.

Within the world of HTTP, there are several ways to validate a user's credentials. Table 6-2 lists the more prevalent ones.

Table 6-2. *Types of Authentication in HTTP*

Type	Description
None	You don't need to know the identity of the user, nor do you need to protect any of the site's or service's resources by applying permissions.
Basic	The caller adds an HTTP authorization header containing a user name and password. Those values are essentially plaintext, using only base64 encoding for simple obfuscation. This generally requires Secure Sockets Layer (SSL) transport security (i.e., an endpoint that exposes an HTTPS address) to protect the plaintext user name and password.
Digest	Provides a fancier method of putting the user name 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 integrated and seamless credential validation. This is 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 user name and password when they visit the intranet.
Public-Key, Certificates	Relies on 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 is an internal, portable, device-based warehousing application for tracking inventory, or maybe a set of iPads used by the sales team. 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 when third-party token issuers are involved (e.g., OpenID, OAuth). This relieves your service of the burden of both storing and verifying a user's credentials. Here's how it works (generally speaking): The caller first verifies the user name and password using a token issuer that your service trusts. Upon successful verification, the token issuer provides the caller with a token. Once the caller has that token, it uses it to call your service. Since your service trusts the issuer that the caller used for credential verification, your service can trust that the token securely identifies the user, and it therefore doesn't have to bother with verifying the user's credentials itself.

In selecting authentication types to support for the task-management service, we can definitely skip the None option because we actually need to identify the caller and enforce permissions.

We can also eliminate Kerberos and Certificates, because the goal is to keep our examples simple and avoid relying on Active Directory. These particular approaches can be overly complex and impractical when dealing with public-facing Internet applications and services.

Between Basic and Digest, Basic is much easier to implement. With Basic authentication, the service application and its callers have to deal only with plain-text credentials. Basic authentication is actually fairly common, and it is viable even in production environments...provided that transport security is used to protect the credentials. Therefore, we will support Basic authentication in the task-management service. However, SSL transport security configuration is a separate topic, outside the scope of this book. We trust that if you decide to implement Basic authentication on your own ASP.NET Web API services, you will enable support for HTTPS.

Finally, we will also support a form of token-based security in the task-management service, because token-based security has become so common these days that it is impossible to ignore (e.g., you've probably heard of OpenID and/or OAuth). It's also a lot easier to implement than it was just a few years ago, thanks to increasing standardization and availability of open-source libraries. Speaking of which, we'll use one of our own libraries to make implementing token-based security as painless as possible.

Authorization

Once the service has securely identified the user, it needs to enforce some basic permissions. The task-management service will have three levels of users, as indicated earlier in Table 6-1.

These days, the concept of *claims* has finally caught 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. 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, the structure can be used for assigning roles. For example, Table 6-3 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).

Table 6-3. *An Example User's Claims*

Claim type	Example claim value
Email	`bob@gmail.com`
UserId	BSmith
Surname	Smith
Givenname	Bob
SID	Bob's security identifier; usually something issued by the system: C73832EE-3191-4DC7-A3D4-25ADDDD5496B
Role	Manager
Role	Senior Worker

Strictly speaking, claims aren't limited to values dealing only with authorization. They do, however, provide a nice structure for indicating the roles a user belongs to, which is of primary interest when it comes to authorization.

Overview of the Authentication and Authorization Process

Before we start coding, let's take a high-level look at what's involved in the authentication and authorization process. Each time a request comes into the task-management service, the following things happen (in this order):

1. A web request arrives that includes an HTTP authorization header containing information about the user and how that information can be verified.
2. The service verifies the user information. Note, however, in the case of token-based authentication, the trusted token issuer has already verified the user information; all the service needs to do is verify that the token was actually issued by the trusted token issuer.
3. The service sets up a security principal object on the current HTTP context that contains the current user's identity and associated claims (e.g., userId, email, firstname, lastname, and roles). Each web request executes in its own HTTP context, so each request will execute within the context of the user's principal.
4. All "downstream" code checks the current context's principal to determine if/how processing is allowed to continue. If processing is not allowed to continue, the service will communicate this to the caller via a response message containing the appropriate HTTP status code (i.e., 401 - Unauthorized).

Now let's get into the implementation so that we can see all of this in action!

■ **Note** As in previous chapters, unless otherwise noted, we implement one public type per file. The file name should match the type name, and the file location should match the namespace name. For example, the `WebApi2Book.Web.Common.Routing.ApiVersionConstraint` class is in a file named "ApiVersionConstraint.cs", which is located in a project folder named "Routing" in the `WebApi2Book.Web.Common` project.

Securing the POST

We implemented the `TasksController` class' `AddTask` action method in Chapter 5, but we did it without securing it in any way. Let's return to that method and get some security around it.

The Authorization Filter

Let's secure the `AddTask` method in the V1 controller (remember, we are finished with the V2 controller) by adding the `Authorize` attribute (i.e., an "authorization filter") to it as follows. Be sure to use the Web API version of the attribute (found in the `System.Web.Http` namespace), and not the MVC version. Also, be sure to add a `using` directive for `WebApi2Book.Common` to satisfy the compiler. Note that we are using the attribute to specify that the user must have a manager role:

```
 [Route("", Name = "AddTaskRoute")]
[HttpPost]
[Authorize(Roles = Constants.RoleNames.Manager)]
```

```
public IHttpActionResult AddTask(HttpRequestMessage requestMessage, NewTask newTask)
{
    var task = _addTaskMaintenanceProcessor.AddTask(newTask);
    var result = new TaskCreatedActionResult(requestMessage, task);
    return result;
}
```

Now, we'll repeat the POST demo from Chapter 5, just to see if anything has changed:

POST Request (abbreviated)

```
POST http://localhost:61589/api/v1/tasks HTTP/1.1
Content-Type: text/json

{"Subject":"Fix something important"}
```

You should see the following response:

POST Response (abbreviated)

```
HTTP/1.1 401 Unauthorized
Content-Type: text/json; charset=utf-8

{"Message":"Authorization has been denied for this request."}
```

Isn't that great? We have secured the POST by simply applying an attribute to the appropriate controller action method! But why was this so easy? The reason is because ASP.NET Web API is doing the heavy lifting for us. First, the framework's message-processing infrastructure detects the presence of the `Authorize` action filter attribute on the `AddTask` method. This causes it to ensure that a security principal containing a manager role has been established on the current HTTP context before invoking the action method. The framework rightly detects that there is no such principal available, and therefore, without ever invoking the target `AddTask` method, it creates an error response (complete with the correct HTTP status code), which it returns to the caller.

So now we've secured our action method, but these POST requests will always fail until 1) they contain information necessary to establish a principal with the manager role, and 2) until we implement the code that actually uses that information to build a principal and associate it with the current context. Let's address this next.

A Message Handler to Support HTTP Basic Authentication

If you review the simplified ASP.NET Web API processing pipeline diagram from the previous chapter (Figure 5-1), you'll notice that message handlers are invoked before filters and controller actions. This makes message handlers well suited to take on the responsibility of building a principal and associating it with the current context. Remember, the principal must be established on the current context before the authorization filter is hit, or else the request will be rejected.

Before we implement our message handler (this first one will support Basic authentication), we need to add the security service to which it delegates some of its principal-building responsibilities. Therefore, add the following types:

IBasicSecurityService Interface

```
namespace WebApi2Book.Web.Api.Security
{
    public interface IBasicSecurityService
    {
        bool SetPrincipal(string username, string password);
    }
}
```

BasicSecurityService Class

```
using System.Security.Claims;
using System.Security.Principal;
using System.Threading;
using System.Web;
using log4net;
using NHibernate;
using WebApi2Book.Common;
using WebApi2Book.Common.Logging;
using WebApi2Book.Data.Entities;
using WebApi2Book.Web.Common;

namespace WebApi2Book.Web.Api.Security
{
    public class BasicSecurityService : IBasicSecurityService
    {
        private readonly ILog _log;

        public BasicSecurityService(ILogManager logManager)
        {
            _log = logManager.GetLog(typeof(BasicSecurityService));
        }

        public virtual ISession Session
        {
            get { return WebContainerManager.Get<ISession>(); }
        }

        ~~/// <summary>~~
        ~~///     An over-simplified method to validate the credentials and set the principal.~~
        ~~/// </summary>~~
        ~~/// <param name="username">The username.</param>~~
        ~~/// <param name="password">Ignored in this implementation.</param>~~
        ~~/// <returns>true if the user was found; otherwise, false</returns>~~
        public bool SetPrincipal(string username, string password)
        {
            var user = GetUser(username);

            IPrincipal principal = null;
            if (user == null || (principal = GetPrincipal(user)) == null)
            {
                _log.DebugFormat("System could not validate user {0}", username);
                return false;
            }

            Thread.CurrentPrincipal = principal;
            if (HttpContext.Current != null)
            {
                HttpContext.Current.User = principal;
            }

            return true;
        }
        public virtual IPrincipal GetPrincipal(User user)
        {
            var identity = new GenericIdentity(user.Username, Constants.SchemeTypes.Basic);

            identity.AddClaim(new Claim(ClaimTypes.GivenName, user.Firstname));
            identity.AddClaim(new Claim(ClaimTypes.Surname, user.Lastname));

            var username = user.Username.ToLowerInvariant();
            switch (username)
            {
                case "bhogg":
                    identity.AddClaim(new Claim(ClaimTypes.Role, Constants.RoleNames.Manager));
                    identity.AddClaim(new Claim(ClaimTypes.Role, Constants.RoleNames.SeniorWorker));
                    identity.AddClaim(new Claim(ClaimTypes.Role, Constants.RoleNames.JuniorWorker));
                    break;
                case "jbob":
                    identity.AddClaim(new Claim(ClaimTypes.Role, Constants.RoleNames.SeniorWorker));
                    identity.AddClaim(new Claim(ClaimTypes.Role, Constants.RoleNames.JuniorWorker));
                    break;
                case "jdoe":
                    identity.AddClaim(new Claim(ClaimTypes.Role, Constants.RoleNames.JuniorWorker));
                    break;
                default:
                    return null;
            }

            return new ClaimsPrincipal(identity);
        }

        public virtual User GetUser(string username)
        {
            username = username.ToLowerInvariant();
            return
                Session.QueryOver<User>().Where(x => x.Username == username).SingleOrDefault();
        }
    }
}
```

Then wire this up so that it can be used as a dependency. This is done by adding the following to the `NinjectConfigurator AddBindings` method:

```
container.Bind<IBasicSecurityService>().To<BasicSecurityService>().InSingletonScope();
```

■ **Note** You'll need to add a `using` directive for `WebApi2Book.Web.Api.Security` to satisfy the compiler.

Let's review. The first thing to note in this security service is that the `ISession` dependency is not constructor-injected. This is because the `BasicSecurityService` is constructed in the application's startup sequence, before the application has prepared an `ISession` instance. (We'll see this when we configure the message handler.) The `Session` property provides the `BasicSecurityService` with "lazy" access to the `ISession` managed by the Ninject container; by the time it accesses it, the ISession is available as a dependency. On a related note, the fact that the BasicSecurityService stores no state itself (other than the ILog instance, which is safe for multithread access) allows us to configure it as a singleton, as you can see in the code snippet just shown.

The next method, SetPrincipal, uses the GetPrincipal method to construct a security principal. If a valid principal can be constructed, it associates the principal with the current thread. This is mostly for legacy purposes (e.g., by convention, some third-party libraries look for a principal on the current thread). More importantly, however, SetPrincipal also places the principal on the current HttpContext. *This is vitally important to do in ASP.NET Web API applications, because while multiple threads may be used to process a single request (and therefore some threads may not automatically have access to the principal via* Thread.CurrentPrincipal*), the current HttpContext and the principal associated with it will be—and will need to be—accessible throughout an entire call.* Fortunately, NuGet security packages for Web API, including the one we're going to use later for token-based security, typically take care of this important detail.

The last method, GetPrincipal, is an extremely simplified approach to constructing a principal from user credentials. In this case, we're only using part of the user's credential, username (used to fetch the User); we're not even verifying the password. Why is this so incredibly simplified? Well, because at this level we're dealing with concerns not unique to ASP.NET Web API. Verifying user credentials at this level should be concerns of a credential management and verification package, like ASP.NET Membership (used the previous edition of this book) or ASP.NET Identity (which is Microsoft's latest approach to membership). As our focus is on ASP.NET Web API, not credential management and verification, we will move on to our custom handler. Go ahead and implement the handler now as follows:

```
using System;
using System.Net;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Web;
using log4net;
using WebApi2Book.Common;
using WebApi2Book.Common.Logging;

namespace WebApi2Book.Web.Api.Security
{
    public class BasicAuthenticationMessageHandler : DelegatingHandler
    {
        public const char AuthorizationHeaderSeparator = ':';
        private const int UsernameIndex = 0;
        private const int PasswordIndex = 1;
        private const int ExpectedCredentialCount = 2;

        private readonly ILog _log;
        private readonly IBasicSecurityService _basicSecurityService;

        public BasicAuthenticationMessageHandler(ILogManager logManager,
            IBasicSecurityService basicSecurityService)
        {
            _basicSecurityService = basicSecurityService;
            _log = logManager.GetLog(typeof (BasicAuthenticationMessageHandler));
        }
```

```
protected override async Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request,
    CancellationToken cancellationToken)
{
    if (HttpContext.Current.User.Identity.IsAuthenticated)
    {
        _log.Debug("Already authenticated; passing on to next handler...");
        return await base.SendAsync(request, cancellationToken);
    }

    if (!CanHandleAuthentication(request))
    {
        _log.Debug("Not a basic auth request; passing on to next handler...");
        return await base.SendAsync(request, cancellationToken);
    }

    bool isAuthenticated;
    try
    {
        isAuthenticated = Authenticate(request);
    }
    catch (Exception e)
    {
        _log.Error("Failure in auth processing", e);
        return CreateUnauthorizedResponse();
    }

    if (isAuthenticated)
    {
        var response = await base.SendAsync(request, cancellationToken);
        return response.StatusCode == HttpStatusCode.Unauthorized ? CreateUnauthorizedResponse():
            response;
    }

    return CreateUnauthorizedResponse();
}

public bool CanHandleAuthentication(HttpRequestMessage request)
{
    return (request.Headers != null
            && request.Headers.Authorization != null
            && request.Headers.Authorization.Scheme.ToLowerInvariant() ==
                    Constants.SchemeTypes.Basic);
}

public bool Authenticate(HttpRequestMessage request)
{
    _log.Debug("Attempting to authenticate...");
```

```
            var authHeader = request.Headers.Authorization;
            if (authHeader == null)
            {
                return false;
            }

            var credentialParts = GetCredentialParts(authHeader);
            if (credentialParts.Length != ExpectedCredentialCount)
            {
                return false;
            }

            return _basicSecurityService.SetPrincipal(credentialParts[UsernameIndex],
                credentialParts[PasswordIndex]);
        }

        public string[] GetCredentialParts(AuthenticationHeaderValue authHeader)
        {
            var encodedCredentials = authHeader.Parameter;
            var credentialBytes = Convert.FromBase64String(encodedCredentials);
            var credentials = Encoding.ASCII.GetString(credentialBytes);
            var credentialParts = credentials.Split(AuthorizationHeaderSeparator);
            return credentialParts;
        }

        public HttpResponseMessage CreateUnauthorizedResponse()
        {
            var response = new HttpResponseMessage(HttpStatusCode.Unauthorized);
            response.Headers.WwwAuthenticate.Add(
                new AuthenticationHeaderValue(Constants.SchemeTypes.Basic));
            return response;
        }
    }
}
```

The first thing to note is that `BasicAuthenticationMessageHandler` derives from `DelegatingHandler`, an ASP.NET Web API base class. The handler overrides the `SendAsync` method. This allows it to pass a request to the next handler in the ASP.NET Web API processing pipeline, in cases where processing is allowed to continue, by calling `base.SendAsync`. It also allows it to return an error response, in cases where processing is not allowed to continue, by calling `CreateUnauthorizedResponse`.

The value added by `CreateUnauthorizedResponse` is seen in the second line of the method, where it adds the `Basic` scheme to the response's `WwwAuthenticate` header. A response with the `Unauthorized` (401) HTTP status code together with the `Basic` scheme in the header will trigger most browsers to prompt for Username and Password.

Next, let's take a look at the following code, which is executed if the user was successfully authenticated:

```
var response = await base.SendAsync(request, cancellationToken);
return response;
```

The first line passes processing along to the next handler and waits for the response. The next line simply returns the response. The rest of the methods are also fairly straightforward:

- `CanHandleAuthentication` examines the request and returns `true` if it contains an HTTP header indicating the `Basic` authorization scheme.
- `Authenticate` uses `GetCredentials` to extract the credentials from the request, and then it delegates the actual work of setting the principal to the security service we implemented earlier.
- `GetCredentials` parses the credentials from the request. The thing to remember here is that the credentials arrive base64-encoded and separated by a delimiter (":").

Now that we've implemented the handler, the last step is to configure it to be added to the application's message-handler pipeline. This is configured during application startup, so let's return to the `WebApiApplication` class and modify it so that it appears as follows:

```
using System.Web;
using System.Web.Http;
using WebApi2Book.Common.Logging;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Web.Api.Security;
using WebApi2Book.Web.Common;

namespace WebApi2Book.Web.Api
{
    public class WebApiApplication : HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);

            RegisterHandlers();

            new AutoMapperConfigurator().Configure(
                WebContainerManager.GetAll<IAutoMapperTypeConfigurator>());
        }

        private void RegisterHandlers()
        {
            var logManager = WebContainerManager.Get<ILogManager>();

            GlobalConfiguration.Configuration.MessageHandlers.Add(
                new BasicAuthenticationMessageHandler(logManager,
                    WebContainerManager.Get<IBasicSecurityService>()));
        }

        protected void Application_Error()
        {
            var exception = Server.GetLastError();
            if (exception != null)
            {
                var log = WebContainerManager.Get<ILogManager>().GetLog(typeof (WebApiApplication));
                log.Error("Unhandled exception.", exception);
            }
        }
    }
}
```

Note the new method, `RegisterHandlers`. We're going to be adding more handlers later, so we figured it would be good to break this configuration out into a separate method. Looking at the implementation, this adds the handler to the `MessageHandlers` collection of the Web API global configuration object, and with this in place, all requests to the task-management service will be intercepted by the `BasicAuthenticationMessageHandler`. For requests decorated with the Basic scheme, the handler will verify the user's credentials and set up a corresponding principal using the trivial implementation in the `BasicSecurityService`. To prove that all of this is working properly, let's now revisit the demo and see this in action!

First, we need to add the `Basic` authentication information to the request message. Here we see the encoded information for `"bhogg"`, who, as you may recall from the `BasicSecurityService` implementation, is a manager. Note the second line; `YmhvZ2c6aWdub3JlZA==` represents bhogg's base64-encoded credentials:

POST Request - Manager (abbreviated)

```
POST http://localhost:61589/api/v1/tasks HTTP/1.1
Authorization: Basic YmhvZ2c6aWdub3JlZA==
Content-Type: text/json

{"Subject":"Fix something important"}
```

Now we'll send the request (we're using Fiddler) and examine the response:

POST Response - Manager (abbreviated)

```
HTTP/1.1 201 Created
Content-Type: text/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":null,"DueDate":null,
"CreatedDate":"2014-05-10T19:02:52.2408621Z","CompletedDate":null,
"Status":{"StatusId":1,"Name":"Not Started","Ordinal":0},"Assignees":[],
"Links":[{"Rel":"self","Href":"http://localhost:61589/api/v1/tasks/17","Method":"GET"}]}
```

Excellent! Our credentials for manager bhogg have been accepted. We've created a task, only this time we've done it securely. Let's make sure the security we've put in place is actually working by sending another request—this time, with the credentials of a user in a junior worker role. This request should be denied, because the `AddTask` controller action requires the user be in the Manager role. Note that `amRvZTppZ25vcmVk` represents the credentials for `jdoe`, the junior worker:

POST Request - Junior Worker (abbreviated)

```
POST http://localhost:61589/api/v1/tasks HTTP/1.1
Authorization: Basic amRvZTppZ25vcmVk
Content-Type: text/json

{"Subject":"Fix something important"}
```

POST Response - Junior Worker (abbreviated)

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: basic
```

Perfect. This demonstrates that we have secured our “Add a Task” scenario; only managers can create a task in our task-management service. The next thing we’ll do is implement the remaining scenarios from Table 6-1 leveraging the security infrastructure we’ve put in place. After that, we’ll show how we can add support for token-based security…without having to modify any of our existing code! This is due to ASP.NET Web API’s excellent extensibility support related to message handlers.

Securing Non-Resource API Operations

In Chapter 3, we designed the API for non-resource API operations. The design is summarized in Table 3-4. What’s missing from the design is the security aspect, so let’s extend the design now by taking security into account, as shown in Table 6-4.

Table 6-4. *A List of Task Status Operations*

URI	Verb	Description	Security
`/api/tasks/123/activations`	POST	Starts, or “activates,” a task; returns the updated task in the response	Requires Senior Worker role
`/api/tasks/123/completions`	POST	Completes a task; returns the updated task in the response	Requires Senior Worker role
`/api/tasks/123/reactivations`	POST	Reopens, or “re-activates,” a task; returns the updated task in the response	Requires Senior Worker role; all reactivations will be audited

Activate a Task

With that as an introduction, let’s start by adding support to activate a task. We’ll follow our usual bottom-up approach of adding dependencies first, and the first dependency we’ll add is a query processor. Remember that the `ITaskByIdQueryProcessor` interface is in the `WebApi2Book.Data` project, while its implementation is in the `WebApi2Book.Data.SqlServer` project (within the QueryProcessors folder).

ITaskByIdQueryProcessor Interface

```
using WebApi2Book.Data.Entities;

namespace WebApi2Book.Data.QueryProcessors
{
    public interface ITaskByIdQueryProcessor
    {
        Task GetTask(long taskId);
    }
}
```

TaskByIdQueryProcessor Class

```
using NHibernate;
using WebApi2Book.Data.Entities;
using WebApi2Book.Data.QueryProcessors;

namespace WebApi2Book.Data.SqlServer.QueryProcessors
{
    public class TaskByIdQueryProcessor : ITaskByIdQueryProcessor
    {
        private readonly ISession _session;

        public TaskByIdQueryProcessor(ISession session)
        {
            _session = session;
        }

        public Task GetTask(long taskId)
        {
            var task = _session.Get<Task>(taskId);
            return task;
        }
    }
}
```

Dependency Configuration (add to NinjectConfigurator.AddBindings)

```
container.Bind<ITaskByIdQueryProcessor>().To<TaskByIdQueryProcessor>().InRequestScope();
```

We introduced the concept of using query processors back in Chapter 5. Basically, query processors are part of a Strategy Pattern implementation to provide access to persistent data. The `TaskByIdQueryProcessor` implementation is fairly trivial, performing a simple "get" against the database by `taskId`.

Our next dependency to add is also a query processor. The query processor we just added is responsible for fetching data, but this one is responsible for updating data. Implement it as follows:

IUpdateTaskStatusQueryProcessor Interface

```
using WebApi2Book.Data.Entities;

namespace WebApi2Book.Data.QueryProcessors
{
    public interface IUpdateTaskStatusQueryProcessor
    {
        void UpdateTaskStatus(Task taskToUpdate, string statusName);
    }
}
```

UpdateTaskStatusQueryProcessor Class

```
using NHibernate;
using WebApi2Book.Data.Entities;
using WebApi2Book.Data.QueryProcessors;

namespace WebApi2Book.Data.SqlServer.QueryProcessors
{
    public class UpdateTaskStatusQueryProcessor : IUpdateTaskStatusQueryProcessor
    {
        private readonly ISession _session;

        public UpdateTaskStatusQueryProcessor(ISession session)
        {
            _session = session;
        }

        public void UpdateTaskStatus(Task taskToUpdate, string statusName)
        {
            var status = _session.QueryOver<Status>().Where(x => x.Name == statusName).SingleOrDefault();

            taskToUpdate.Status = status;

            _session.SaveOrUpdate(taskToUpdate);
        }
    }
}
```

Dependency Configuration (add to NinjectConfigurator.AddBindings)

```
container.Bind<IUpdateTaskStatusQueryProcessor>().To<UpdateTaskStatusQueryProcessor>()
        .InRequestScope();
```

The `UpdateTaskStatusQueryProcessor` implementation is similarly unremarkable. It's just finding the appropriate status object and associating it with the task.

The next dependency is slightly more interesting. It performs all of the "business logic" required to activate a task. Implement it as follows:

IStartTaskWorkflowProcessor Interface

```
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public interface IStartTaskWorkflowProcessor
    {
        Task StartTask(long taskId);
    }
}
```

StartTaskWorkflowProcessor Class

```
using WebApi2Book.Common;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Data.Exceptions;
using WebApi2Book.Data.QueryProcessors;
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public class StartTaskWorkflowProcessor : IStartTaskWorkflowProcessor
    {
        private readonly IAutoMapper _autoMapper;
        private readonly ITaskByIdQueryProcessor _taskByIdQueryProcessor;
        private readonly IDateTime _dateTime;
        private readonly IUpdateTaskStatusQueryProcessor _updateTaskStatusQueryProcessor;

        public StartTaskWorkflowProcessor(ITaskByIdQueryProcessor taskByIdQueryProcessor,
            IUpdateTaskStatusQueryProcessor updateTaskStatusQueryProcessor, IAutoMapper autoMapper,
             IDateTime dateTime)
        {
            _taskByIdQueryProcessor = taskByIdQueryProcessor;
            _updateTaskStatusQueryProcessor = updateTaskStatusQueryProcessor;
            _autoMapper = autoMapper;
            _dateTime = dateTime;
        }

        public Task StartTask(long taskId)
        {
            var taskEntity = _taskByIdQueryProcessor.GetTask(taskId);
            if (taskEntity == null)
            {
                throw new RootObjectNotFoundException("Task not found");
            }

            // Simulate some workflow logic...
            if (taskEntity.Status.Name != "Not Started")
            {
                throw new BusinessRuleViolationException(
                    "Incorrect task status. Expected status of 'Not Started'.");
            }

            taskEntity.StartDate = _dateTime.UtcNow;
            _updateTaskStatusQueryProcessor.UpdateTaskStatus(taskEntity, "In Progress");

            var task = _autoMapper.Map<Task>(taskEntity);

            return task;
        }
    }
}
```

Dependency Configuration (add to NinjectConfigurator.AddBindings)

```
container.Bind<IStartTaskWorkflowProcessor>().To<StartTaskWorkflowProcessor>().InRequestScope();
```

We'll explain this next, but first we have to add one more dependency to satisfy the compiler: `BusinessRuleViolationException`. Instances of this trivial exception type are thrown to indicate an attempted violation of the business logic. Implement it as follows:

BusinessRuleViolationException Class

```
using System;

namespace WebApi2Book.Common
{
    public class BusinessRuleViolationException : Exception
    {
        public BusinessRuleViolationException(string incorrectTaskStatus) :
base(incorrectTaskStatus)
        {
        }
    }
}
```

Now that we've made the compiler happy, let's review the `StartTaskWorkflowProcessor` class' `StartTask` method. It begins by delegating to `ITaskByIdQueryProcessor` to find a `Task` entity with the specified `taskId`. An instance of the `RootObjectNotFoundException` class, which was introduced in Chapter 5, is thrown if no such `Task` can be found.

Next, we encounter some "business logic", which enforces a business rule requiring a task to have a status of `Not Started` in order to be activated. If that condition is satisfied, the `ITaskByIdQueryProcessor` sets the task's `StartDate`, and then delegates the job of actually updating the `Status` to the `IUpdateTaskStatusQueryProcessor`. Last, the injected `IAutoMapper` dependency converts the task from an entity representation to a service model representation, which is then returned to the invoker of the `StartTask` method.

And now it's finally time to implement that invoker of the `StartTask` method, which is the `TaskWorkflowController.` (Remember, these are "conceptual resources," as we discussed in Chapter 3, which is why this method is not on the `TasksController`.) Implement it as follows:

```
using System.Web.Http;
using WebApi2Book.Common;
using WebApi2Book.Web.Api.MaintenanceProcessing;
using WebApi2Book.Web.Api.Models;
using WebApi2Book.Web.Common;
using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api.Controllers.V1
{
    [ApiVersion1RoutePrefix("")]
    [UnitOfWorkActionFilter]
    public class TaskWorkflowController : ApiController
    {
        private readonly IStartTaskWorkflowProcessor _startTaskWorkflowProcessor;
        public TaskWorkflowController(IStartTaskWorkflowProcessor startTaskWorkflowProcessor)
        {
            _startTaskWorkflowProcessor = startTaskWorkflowProcessor;
        }

        [HttpPost]
        [Authorize(Roles = Constants.RoleNames.SeniorWorker)]
        [Route("tasks/{taskId:long}/activations", Name = "StartTaskRoute")]
        public Task StartTask(long taskId)
        {
            var task = _startTaskWorkflowProcessor.StartTask(taskId);
            return task;
        }
    }
}
```

And that's it; we have implemented the ability to activate a task! Note that we've leveraged several attributes with this implementation. However, we've discussed all of these before, so we will press onward.

Before we can rightly claim victory and move on to the next scenario, though, we should prove that what we've implemented actually works. We'll send the following request to activate task #17, which we created at the end of the previous section. *Be sure to substitute the actual number of the task you created if it wasn't 17.* Note that we're providing bhogg's credentials to ensure that the request is authorized (because he's a member of the Senior Worker role):

Activate Task Request (abbreviated)

```
POST http://localhost:61589/api/v1/tasks/17/activations HTTP/1.1
Authorization: Basic YmhvZ2c6aWdub3JlZA==
```

Activate Task Response (abbreviated)

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34.2373052Z",
"DueDate":null,"CreatedDate":"2014-05-10T19:02:52","CompletedDate":null,
"Status":{"StatusId":2,"Name":"In Progress","Ordinal":1},"Assignees":[],"Links":[]}
```

This is correct so far. The task is now `In Progress`, and we have a non-null value for `StartDate`. Now let's test our business logic requiring tasks to have a `Not Started` status in order to be activated. To do so, send the request again... and the result is

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json; charset=utf-8

{"Message":"Incorrect task status. Expected status of 'Not Started'."}
```

Well, the message looks OK, but the status code is incorrect. We should be returning a status code of 402 to indicate a custom business rule violation, not 500, which indicates a server error. Fortunately, we have the `GlobalExceptionHandler`, and this status-code translation is a perfect job for it. Add the highlighted code to its `Handle` method, add a `using` directive for `WebApi2Book.Common`, and then retry the request:

```
...
if (exception is ChildObjectNotFoundException)
{
    context.Result = new SimpleErrorResult(context.Request, HttpStatusCode.Conflict, exception.Message);
    return;
}

if (exception is BusinessRuleViolationException)
{
    context.Result = new SimpleErrorResult(context.Request, HttpStatusCode.PaymentRequired,
        exception.Message);
    return;
}

context.Result = new SimpleErrorResult(context.Request, HttpStatusCode.InternalServerError,
    exception.Message);
...
```

You should see something similar to the following (note the change in the response code):

```
HTTP/1.1 402 Payment Required
Content-Type: application/json; charset=utf-8

{"Message":"Incorrect task status. Expected status of 'Not Started'."}
```

This is exactly what we wanted to see! Now we can rightly claim victory and move on to the next scenario.

Complete a Task

This section will go quickly, because we've already implemented most of the dependencies we need to complete a task. The first, and only, new dependency we need to add is `ICompleteTaskWorkflowProcessor`. Go ahead and implement it now:

ICompleteTaskWorkflowProcessor Interface

```
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public interface ICompleteTaskWorkflowProcessor
    {
        Task CompleteTask(long taskId);
    }
}
```

CompleteTaskWorkflowProcessor Class

```
using WebApi2Book.Common;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Data.Exceptions;
using WebApi2Book.Data.QueryProcessors;
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public class CompleteTaskWorkflowProcessor : ICompleteTaskWorkflowProcessor
    {
        private readonly IAutoMapper _autoMapper;
        private readonly ITaskByIdQueryProcessor _taskByIdQueryProcessor;
        private readonly IDateTime _dateTime;
        private readonly IUpdateTaskStatusQueryProcessor _updateTaskStatusQueryProcessor;

        public CompleteTaskWorkflowProcessor(ITaskByIdQueryProcessor taskByIdQueryProcessor,
            IUpdateTaskStatusQueryProcessor updateTaskStatusQueryProcessor, IAutoMapper autoMapper,
            IDateTime dateTime)
        {
            _taskByIdQueryProcessor = taskByIdQueryProcessor;
            _updateTaskStatusQueryProcessor = updateTaskStatusQueryProcessor;
            _autoMapper = autoMapper;
            _dateTime = dateTime;
        }

        public Task CompleteTask(long taskId)
        {
            var taskEntity = _taskByIdQueryProcessor.GetTask(taskId);
            if (taskEntity == null)
            {
                throw new RootObjectNotFoundException("Task not found");
            }

            // Simulate some workflow logic...
            if (taskEntity.Status.Name != "In Progress")
            {
                throw new BusinessRuleViolationException(
                    "Incorrect task status. Expected status of 'In Progress'.");
            }

            taskEntity.CompletedDate = _dateTime.UtcNow;
            _updateTaskStatusQueryProcessor.UpdateTaskStatus(taskEntity, "Completed");

            var task = _autoMapper.Map<Task>(taskEntity);

            return task;
        }
    }
}
```

Dependency Configuration (add to NinjectConfigurator.AddBindings)

```
container.Bind<ICompleteTaskWorkflowProcessor>().To<CompleteTaskWorkflowProcessor>()
    .InRequestScope();
```

This class is similar to the StartTaskWorkflowProcessor, only this time we are requiring a status of In Progress in order to complete the processing, and we are updating the task's CompletedDate rather than the StartDate.

Now, let's hook it up to the controller, which should appear as follows:

```
using System.Web.Http;
using WebApi2Book.Common;
using WebApi2Book.Web.Api.MaintenanceProcessing;
using WebApi2Book.Web.Api.Models;
using WebApi2Book.Web.Common;
using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api.Controllers.V1
{
    [ApiVersion1RoutePrefix("")]
    [UnitOfWorkActionFilter]
    [Authorize(Roles = Constants.RoleNames.SeniorWorker)]
    public class TaskWorkflowController : ApiController
    {
        private readonly IStartTaskWorkflowProcessor _startTaskWorkflowProcessor;
        private readonly ICompleteTaskWorkflowProcessor _completeTaskWorkflowProcessor;

        public TaskWorkflowController(IStartTaskWorkflowProcessor startTaskWorkflowProcessor,
            ICompleteTaskWorkflowProcessor completeTaskWorkflowProcessor)
        {
            _startTaskWorkflowProcessor = startTaskWorkflowProcessor;
            _completeTaskWorkflowProcessor = completeTaskWorkflowProcessor;
        }

        [HttpPost]
        [Route("tasks/{taskId:long}/activations", Name = "StartTaskRoute")]
        public Task StartTask(long taskId)
        {
            var task = _startTaskWorkflowProcessor.StartTask(taskId);
            return task;
        }

        [HttpPost]
        [Route("tasks/{taskId:long}/completions", Name = "CompleteTaskRoute")]
        public Task CompleteTask(long taskId)
        {
            var task = _completeTaskWorkflowProcessor.CompleteTask(taskId);
            return task;
        }
    }
}
```

As you can see, the controller is still quite simple. However, note that we've moved the `Authorize` attribute from the `StartTask` method and placed it on the controller class itself. As a result, we've broadened its scope of influence. Instead of restricting `StartTask` to users with a Senior Worker role, it is now requiring a Senior Worker role for every action method in the controller. This is exactly what we want, and it sure cuts down on clutter that would otherwise be introduced by copy|paste. We could even apply the `Authorize` attribute at the global configuration level—ASP.NET Web API supports this kind of global application of attributes—but we won't because it doesn't meet our security requirements from a "business" perspective. Still, it's good to know the capability exists.

Before wrapping up this section, let's send a couple of requests to ensure everything is working properly. First, let's close task #17:

Complete Task Request (abbreviated)

```
POST http://localhost:61589/api/v1/tasks/17/completions HTTP/1.1
Authorization: Basic YmhvZ2c6aWdub3JlZA==
```

Complete Task Response (abbreviated)

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34","DueDate":null,
"CreatedDate":"2014-05-10T19:02:52","CompletedDate":"2014-05-13T02:13:08.9855782Z",
"Status":{"StatusId":3,"Name":"Completed","Ordinal":2},"Assignees":[],"Links":[]}
```

So far, so good. We see the Status and CompletedDate are being assigned properly. Now let's retry the message so that we can see if our business rule is being enforced:

```
HTTP/1.1 402 Payment Required
Content-Type: application/json; charset=utf-8

{"Message":"Incorrect task status. Expected status of 'In Progress'."}
```

It is, indeed! Just what we wanted to see. Now on to the final scenario in this section.

Reactivate a Task

This section will be much like the previous one. We'll throw in a little twist, though: in this section, we're going to audit task reactivations using a custom async filter. With the release of ASP.NET Web API 2.1, async filter implementation is greatly simplified because the framework now provides virtual `On*Async` methods to override. We'll show just how simple it is…after we implement the controller method.

The first dependency we need to add is `IReactivateTaskWorkflowProcessor`. Go ahead and implement it now:

IReactivateTaskWorkflowProcessor Interface

```
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public interface IReactivateTaskWorkflowProcessor
    {
        Task ReactivateTask(long taskId);
    }
}
```

ReactivateTaskWorkflowProcessor Class

```
using WebApi2Book.Common;
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Data.Exceptions;
using WebApi2Book.Data.QueryProcessors;
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing
{
    public class ReactivateTaskWorkflowProcessor : IReactivateTaskWorkflowProcessor
    {
        private readonly IAutoMapper _autoMapper;
        private readonly ITaskByIdQueryProcessor _taskByIdQueryProcessor;
        private readonly IUpdateTaskStatusQueryProcessor _updateTaskStatusQueryProcessor;

        public ReactivateTaskWorkflowProcessor(ITaskByIdQueryProcessor taskByIdQueryProcessor,
            IUpdateTaskStatusQueryProcessor updateTaskStatusQueryProcessor, IAutoMapper autoMapper)
        {
            _taskByIdQueryProcessor = taskByIdQueryProcessor;
            _updateTaskStatusQueryProcessor = updateTaskStatusQueryProcessor;
            _autoMapper = autoMapper;
        }

        public Task ReactivateTask(long taskId)
        {
            var taskEntity = _taskByIdQueryProcessor.GetTask(taskId);
            if (taskEntity == null)
            {
                throw new RootObjectNotFoundException("Task not found");
            }

            // Simulate some workflow logic...
            if (taskEntity.Status.Name != "Completed")
            {
                throw new BusinessRuleViolationException(
                    "Incorrect task status. Expected status of 'Completed'.");
            }

            taskEntity.CompletedDate = null;
            _updateTaskStatusQueryProcessor.UpdateTaskStatus(taskEntity, "In Progress");

            var task = _autoMapper.Map<Task>(taskEntity);

            return task;
        }
    }
}
```

Dependency Configuration (add to NinjectConfigurator.AddBindings)

```
container.Bind<IReactivateTaskWorkflowProcessor>().To<ReactivateTaskWorkflowProcessor>()
    .InRequestScope();
```

This class is similar to the CompleteTaskWorkflowProcessor, only this time we are requiring a status of Completed in order to complete the processing, and we are updating the task's CompletedDate by resetting it to null.

Now, let's hook it up to the controller, which should appear as follows:

```
using System.Web.Http;
using WebApi2Book.Common;
using WebApi2Book.Web.Api.MaintenanceProcessing;
using WebApi2Book.Web.Api.Models;
using WebApi2Book.Web.Common;
using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api.Controllers.V1
{
    [ApiVersion1RoutePrefix("")]
    [UnitOfWorkActionFilter]
    [Authorize(Roles = Constants.RoleNames.SeniorWorker)]
    public class TaskWorkflowController : ApiController
    {
        private readonly IStartTaskWorkflowProcessor _startTaskWorkflowProcessor;
        private readonly ICompleteTaskWorkflowProcessor _completeTaskWorkflowProcessor;
        private readonly IReactivateTaskWorkflowProcessor _reactivateTaskWorkflowProcessor;

        public TaskWorkflowController(IStartTaskWorkflowProcessor startTaskWorkflowProcessor,
            ICompleteTaskWorkflowProcessor completeTaskWorkflowProcessor,
            IReactivateTaskWorkflowProcessor reactivateTaskWorkflowProcessor)
        {
            _startTaskWorkflowProcessor = startTaskWorkflowProcessor;
            _completeTaskWorkflowProcessor = completeTaskWorkflowProcessor;
            _reactivateTaskWorkflowProcessor = reactivateTaskWorkflowProcessor;
        }

        [HttpPost]
        [Route("tasks/{taskId:long}/activations", Name = "StartTaskRoute")]
        public Task StartTask(long taskId)
        {
            var task = _startTaskWorkflowProcessor.StartTask(taskId);
            return task;
        }

        [HttpPost]
        [Route("tasks/{taskId:long}/completions", Name = "CompleteTaskRoute")]
        public Task CompleteTask(long taskId)
        {
            var task = _completeTaskWorkflowProcessor.CompleteTask(taskId);
            return task;
        }
        [HttpPost]
        [Route("tasks/{taskId:long}/reactivations", Name = "ReactivateTaskRoute")]
        public Task ReactivateTask(long taskId)
        {
            var task = _reactivateTaskWorkflowProcessor.ReactivateTask(taskId);
            return task;
        }
    }
}
```

Again, the controller is still quite simple. Let's send a couple of requests to ensure everything is working properly. First, let's reactivate task #17 using bhogg's credentials:

Reactivate Task Request (abbreviated)

```
POST http://localhost:61589/api/v1/tasks/17/reactivations HTTP/1.1
Authorization: Basic YmhvZ2c6aWdub3JlZA==
```

Reactivate Task Response (abbreviated)

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34","DueDate":null,
"CreatedDate":"2014-05-10T19:02:52","CompletedDate":null,
"Status":{"StatusId":2,"Name":"In Progress","Ordinal":1},"Assignees":[],"Links":[]}
```

So far, so good. We see that Status and CompletedDate are being updated properly. Now let's retry the message so that we can see if our business rule is being enforced:

```
HTTP/1.1 402 Payment Required
Content-Type: application/json; charset=utf-8

{"Message":"Incorrect task status. Expected status of 'Completed'."}
```

Perfect! It's not allowing us to reactivate a task that is already active, and the response looks correct.

Auditing

So now let's get to the auditing. The first thing to do is implement the custom attribute (implement it as follows):

```
using System.Threading;
using System.Threading.Tasks;
using System.Web.Http.Controllers;
using System.Web.Http.Filters;
using log4net;
using WebApi2Book.Common.Logging;
using WebApi2Book.Common.Security;
```

```
namespace WebApi2Book.Web.Common.Security
{
    public class UserAuditAttribute : ActionFilterAttribute
    {
        private readonly ILog _log;
        private readonly IUserSession _userSession;

        public UserAuditAttribute()
            : this(WebContainerManager.Get<ILogManager>(), WebContainerManager.Get<IUserSession>())
        {
        }

        public UserAuditAttribute(ILogManager logManager, IUserSession userSession)
        {
            _userSession = userSession;
            _log = logManager.GetLog(typeof (UserAuditAttribute));
        }

        public override bool AllowMultiple
        {
            get { return false; }
        }

        public override Task OnActionExecutingAsync(HttpActionContext actionContext,
            CancellationToken cancellationToken)
        {
            _log.Debug("Starting execution...");
            var userName = _userSession.Username;
            return Task.Run(() => AuditCurrentUser(userName), cancellationToken);
        }

        public void AuditCurrentUser(string username)
        {
            // Simulate long auditing process
            _log.InfoFormat("Action being executed by user={0}", username);
            Thread.Sleep(3000);
        }

        public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)
        {
            _log.InfoFormat("Action executed by user={0}", _userSession.Username);
        }
    }
}
```

And now apply the attribute to the controller's ReactivateTask method:

```
[HttpPost]
[UserAudit]
[Route("tasks/{taskId:long}/reactivations", Name = "ReactivateTaskRoute")]
public Task ReactivateTask(long taskId)
{
    var task = _reactivateTaskWorkflowProcessor.ReactivateTask(taskId);
    return task;
}
```

■ **Note** You'll also need to add a `using` directive for `WebApi2Book.Web.Common.Security` to satisfy the compiler.

Although most of its implementation consists of writing debug messages and sleeping, this attribute demonstrates the ability to do something useful in terms of security; specifically, the ability to noninvasively audit user actions. By simply applying this attribute to a controller action method, to an entire controller, or even to the global configuration, we have added auditing.

The attribute's implementation is straightforward, with a couple of items to note: First, in an actual production application, you will probably be writing auditing information to a database, not to a log file. Second, notice how we capture the user name in `OnActionExecutingAsync` prior to performing the actual audit. This is because the `HttpContext.Current.User` object that our `IUserSession` implementation relies upon is not available in the `AuditCurrentUser` method, so we need to first capture this information from `IUserSession` and then pass it into `AuditCurrentUser`.

Now let's see the auditing attribute in action. At this point, task #17 needs to be "completed" before we can reactivate it. So go ahead and send a completion request to get it into the correct state for reactivating:

```
POST http://localhost:61589/api/v1/tasks/17/completions HTTP/1.1
Authorization: Basic YmhvZ2c6aWdub3JlZA==
```

Assuming this was successful, send a request to reactivate the task:

```
POST http://localhost:61589/api/v1/tasks/17/reactivations HTTP/1.1
Authorization: Basic YmhvZ2c6aWdub3JlZA==
```

Nothing new to see here; you should have received a response similar to the response from the first reactivation, shown earlier. But now let's look in the application's log file to verify that we are, indeed, auditing the reactivation. Amidst the many lines of `SimpleTraceWriter`-generated information, you should see something similar to the following:

```
2014-05-13 12:52:38,990 DEBUG [81] WebApi2Book.Web.Common.Security.UserAuditAttribute
  - Starting execution...
2014-05-13 12:52:38,990 INFO  [82] WebApi2Book.Web.Common.Security.UserAuditAttribute
  - Action being executed by user=bhogg
2014-05-13 12:52:42,068 INFO  [82] WebApi2Book.Web.Common.Security.UserAuditAttribute
  - Action executed by user=bhogg
```

And that's it. By leveraging ASP.NET Web API's improved async filter support, we have easily and noninvasively provided auditing for task reactivation. Before we conclude this section, note how processing switched from thread #81 to thread #82. This helps illustrate why we always need to ensure that the principal is associated with the `HttpContext`, as mentioned earlier in the "A Message Handler to Support HTTP Basic Authentication" section, and not just the thread principal.

GET a Task

The last scenario we'll implement (described in Table 6-5) in this chapter involves retrieving data for a particular task.

Table 6-5. *Get a Task*

URI	Verb	Description
`/api/tasks/123`	GET	Gets the details for a single task

We'll restrict the operation itself to users who are at least junior workers through the use of an authorization filter, and we'll also implement a message handler to remove sensitive data from the response for users who are not members of the Senior Worker role. So much to do—let's get started!

Keeping with our bottom-up, thin-controller approach, we'll begin by adding the class that the controller delegates its work to:

ITaskByIdInquiryProcessor Interface

```
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.InquiryProcessing
{
    public interface ITaskByIdInquiryProcessor
    {
        Task GetTask(long taskId);
    }
}
```

TaskByIdInquiryProcessor Class

```
using WebApi2Book.Common.TypeMapping;
using WebApi2Book.Data.QueryProcessors;
using WebApi2Book.Data.Exceptions;
using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.InquiryProcessing
{
    public class TaskByIdInquiryProcessor : ITaskByIdInquiryProcessor
    {
        private readonly IAutoMapper _autoMapper;
        private readonly ITaskByIdQueryProcessor _queryProcessor;

        public TaskByIdInquiryProcessor(ITaskByIdQueryProcessor queryProcessor,
            IAutoMapper autoMapper)
        {
            _queryProcessor = queryProcessor;
            _autoMapper = autoMapper;
        }

        public Task GetTask(long taskId)
        {
            var taskEntity = _queryProcessor.GetTask(taskId);
            if (taskEntity == null)
            {
                throw new RootObjectNotFoundException("Task not found");
            }

            var task = _autoMapper.Map<Task>(taskEntity);

            return task;
        }
    }
}
```

Dependency Configuration (add to NinjectConfigurator.AddBindings)

```
container.Bind<ITaskByIdInquiryProcessor>().To<TaskByIdInquiryProcessor>().InRequestScope();
```

■ **Note** You'll also need to add a `using` directive for `WebApi2Book.Web.Api.InquiryProcessing` to satisfy the compiler.

It doesn't get much easier than that. `GetTask` uses the `ITaskByIdQueryProcessor` (which we added earlier in the "Activate a Task" section) to fetch the entity. Then it uses the `IAutoMapper` to transfer the data to a service model object that can be returned in the response.

To complete the basic scenario, add the new `GetTask` method to the `TasksController`, which should now appear as follows:

```
using System.Net.Http;
using System.Web.Http;
using WebApi2Book.Common;
using WebApi2Book.Web.Api.InquiryProcessing;
using WebApi2Book.Web.Api.MaintenanceProcessing;
using WebApi2Book.Web.Api.Models;
using WebApi2Book.Web.Common;
using WebApi2Book.Web.Common.Routing;

namespace WebApi2Book.Web.Api.Controllers.V1
{
    [ApiVersion1RoutePrefix("tasks")]
    [UnitOfWorkActionFilter]
    public class TasksController : ApiController
    {
        private readonly IAddTaskMaintenanceProcessor _addTaskMaintenanceProcessor;
        private readonly ITaskByIdInquiryProcessor _taskByIdInquiryProcessor;

        public TasksController(IAddTaskMaintenanceProcessor addTaskMaintenanceProcessor,
            ITaskByIdInquiryProcessor taskByIdInquiryProcessor)
        {
            _addTaskMaintenanceProcessor = addTaskMaintenanceProcessor;
            _taskByIdInquiryProcessor = taskByIdInquiryProcessor;
        }
        [Route("", Name = "AddTaskRoute")]
        [HttpPost]
        [Authorize(Roles = Constants.RoleNames.Manager)]
        public IHttpActionResult AddTask(HttpRequestMessage requestMessage, NewTask newTask)
        {
            var task = _addTaskMaintenanceProcessor.AddTask(newTask);
            var result = new TaskCreatedActionResult(requestMessage, task);
            return result;
        }

        [Route("{id:long}", Name = "GetTaskRoute")]
                public Task GetTask(long id)
                {
                    var task = _taskByIdInquiryProcessor.GetTask(id);
                    return task;
                }
    }
}
```

Also add an authorization filter to the controller to ensure that users are members of the Junior Worker role as a minimum requirement to perform any task operation:

```
[ApiVersion1RoutePrefix("tasks")]
[UnitOfWorkActionFilter]
[Authorize(Roles = Constants.RoleNames.JuniorWorker)]
public class TasksController : ApiController
```

At this point, `[Authorize(Roles = Constants.RoleNames.Manager)]` is still decorating the `AddTask` method, because adding a task requires a manager role. This illustrates how the authorization requirements of the filter with the narrower scope (the one on the method) are combined with the requirements of the one with the broader scope (the one on the class). As a result, the user has to satisfy all of the authorization requirements in the entire chain.

Now let's send a request to prove that this is working properly:

Get Task Request (abbreviated)

```
GET http://localhost:61589/api/v1/tasks/17 HTTP/1.1
Authorization: Basic YmhvZ2c6aWdub3JlZA==
```

Get Task Response (abbreviated)

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34","DueDate":null,
"CreatedDate":"2014-05-10T19:02:52","CompletedDate":null,
"Status":{"StatusId":2,"Name":"In Progress","Ordinal":1},"Assignees":[],"Links":[]}
```

Yes, it looks like our Get operation is working properly, so we will move on to implementing a message handler that can conditionally remove sensitive data from the response. Before we do that, though, we need to add some members to the Task service model class for the message handler to use to remove sensitive data. Add the following members to WebApi2Book.Web.Api.Models.Task:

```
private bool _shouldSerializeAssignees;

public void SetShouldSerializeAssignees(bool shouldSerialize)
{
    _shouldSerializeAssignees = shouldSerialize;
}

public bool ShouldSerializeAssignees()
{
    return _shouldSerializeAssignees;
}
```

By convention, ASP.NET Web API uses reflection to call ShouldSerialize* methods to determine if specific public properties should be serialized. We leverage this behavior by pairing ShouldSerializeAssignees with the SetShouldSerializeAssignees method to control the serialization of assignees. In doing so, we have provided a mechanism for the message handler to dynamically control whether a task's assignees should be serialized.

Now it's time to implement the message handler, so implement it as follows:

```
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
using log4net;
using WebApi2Book.Common;
using WebApi2Book.Common.Logging;
using WebApi2Book.Common.Security;
using Task = WebApi2Book.Web.Api.Models.Task;

namespace WebApi2Book.Web.Api.Security
{
    public class TaskDataSecurityMessageHandler : DelegatingHandler
    {
        private readonly ILog _log;
        private readonly IUserSession _userSession;

        public TaskDataSecurityMessageHandler(ILogManager logManager, IUserSession userSession)
        {
            _userSession = userSession;
            _log = logManager.GetLog(typeof (TaskDataSecurityMessageHandler));
        }

        protected override async Task<HttpResponseMessage> SendAsync(
            HttpRequestMessage request,
            CancellationToken cancellationToken)
        {
            var response = await base.SendAsync(request, cancellationToken);
            if (CanHandleResponse(response))
            {
                ApplySecurityToResponseData((ObjectContent) response.Content);
            }

            return response;
        }

        public bool CanHandleResponse(HttpResponseMessage response)
        {
            var objectContent = response.Content as ObjectContent;
            var canHandleResponse = objectContent != null && objectContent.ObjectType == typeof (Task);
            return canHandleResponse;
        }

        public void ApplySecurityToResponseData(ObjectContent responseObjectContent)
        {
            var removeSensitiveData = !_userSession.IsInRole(Constants.RoleNames.SeniorWorker);

            if (removeSensitiveData)
            {
                _log.DebugFormat("Applying security data masking for user {0}", _userSession.Username);
            }

            ((Task) responseObjectContent.Value).SetShouldSerializeAssignees
(!removeSensitiveData);        }
    }
}
```

We'll review the handler code shortly, but first we need to write the code to configure it at run time. The handler is configured during application startup, so let's return to the `WebApiApplication` class and modify the `RegisterHandlers` method so that it appears as follows:

```
private void RegisterHandlers()
{
    var logManager = WebContainerManager.Get<ILogManager>();
    var userSession = WebContainerManager.Get<IUserSession>();

    GlobalConfiguration.Configuration.MessageHandlers.Add(
        new BasicAuthenticationMessageHandler(logManager,
            WebContainerManager.Get<IBasicSecurityService>()));

    GlobalConfiguration.Configuration.MessageHandlers.Add(
        new TaskDataSecurityMessageHandler(logManager, userSession));
}
```

■ **Note** You'll also need to add a `using` directive for `WebApi2Book.Common.Security` to satisfy the compiler.

Looking back at the TaskDataSecurityMessageHandler implementation, we see that it, like BasicAuthenticationMessageHandler, derives from the ASP.NET Web API's DelegatingHandler base class. As with BasicAuthenticationMessageHandler, it overrides SendAsync. However, this time the implementation passes the request on down the chain with a call to base.SendAsync without inspecting it first. That's because this handler is interested only in the response, and it's going to act upon the response only if CanHandleResponse returns true, indicating that the response contains a Task service model object.

If CanHandleResponse indicates that the response contains task data, SendAsync invokes ApplySecurityToResponseData. It is here that the assignees are removed from the response if the user does not have a Senior Worker role. ApplySecurityToResponseData also logs the removal, for debugging purposes.

With this is in place, let's do a demo to make sure this sensitive data-removal functionality is working. The demo won't be terribly dramatic, because at this point we haven't associated any users (or "assignees") with any tasks. However, we can at least verify that the Assignees property is being removed from the response for users who lack a Senior Worker role.

First, we'll test the functionality with bhogg's credentials. We can do this by repeating the request we sent at the beginning of this section. You should see something similar to the following:

Get Task Request (abbreviated)

```
GET http://localhost:61589/api/v1/tasks/17 HTTP/1.1
Authorization: Basic YmhvZ2c6aWdub3JlZA==
```

Get Task Response (abbreviated)

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34","DueDate":null,
"CreatedDate":"2014-05-10T19:02:52","CompletedDate":null,
"Status":{"StatusId":2,"Name":"In Progress","Ordinal":1},"Assignees":[],"Links":[]}
```

Note that the Assignees property appears in the response, as expected. Now repeat the test, this time with jdoe's credentials:

Get Task Request (abbreviated)

```
GET http://localhost:61589/api/v1/tasks/17 HTTP/1.1
Authorization: Basic amRvZTppZ25vcmVk
```

Get Task Response (abbreviated)

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"TaskId":17,"Subject":"Fix something important","StartDate":"2014-05-13T00:52:34","DueDate":null,
"CreatedDate":"2014-05-10T19:02:52","CompletedDate":null,
"Status":{"StatusId":2,"Name":"In Progress","Ordinal":1},"Links":[]}
```

Correct! User jdoe is only a junior worker, so the handler stripped the Assignees property from the response. Our security is working perfectly!

In the next section, we will bring token-based security into the picture and demonstrate how noninvasive the process of adding new authentication schemes can be with ASP.NET Web API.

Applying Token-Based Security

In the previous section, we demonstrated two important concepts. One, utilizing the extensibility of ASP.NET Web API to build a message handler that deals with a service-level concern (i.e., authentication). And two, the simple strategy and mechanics behind HTTP Basic authentication. Now we'd like to cover the basics of token-based security, as well as securing your ASP.NET Web API service with a token-based authentication library.

Token Basics

The most important concept to understand when it comes to token security is this: your service doesn't validate the user's credentials. In fact, your service doesn't even receive the user's credentials, at least in the traditional sense. In place of the user's user name and password, you receive a token that was created and signed by a trusted issuer. And because your service trusts the token issuer, you can merely accept the token as-is, without having to separately validate any credentials from the caller. In the context of securing your service, a token

- Identifies the user
- Contains contents that can be trusted (signed)
- Can be used in place of credentials (user name and password)
- Can contain other attributes

Your driver's license is an example of a "token" you use on a regular basis. Similar to the preceding list, a driver's license

- Identifies the holder
- Is generally trusted
- Requires no other secrets or proof
- Contains other attributes (or, "claims") of the holder

First, your driver's license is used to identify you. Pretty simple! Next, most people checking your driver's license will trust that it is valid. This is because the license is issued by a government entity and is usually signed in some fashion (e.g., water mark). And because it is trusted, the person checking your license doesn't need to ask for a password or PIN or any other proof of identity. And finally, your license includes other attributes beyond just your identity. In application security vernacular, the identity and other attributes are often referred to as "claims" (as discussed at the beginning of this chapter). That is, you are claiming various things about yourself with your license. You can see a sample driver's license in Figure 6-1.

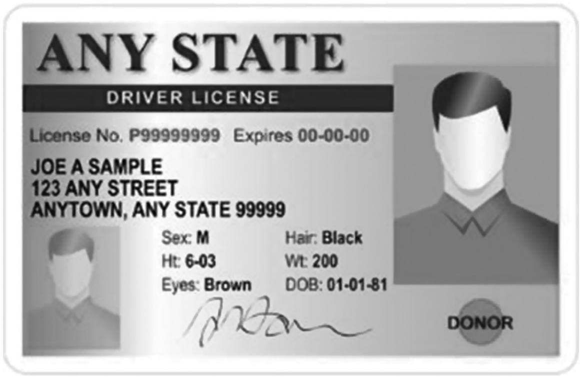

Figure 6-1. *Sample driver's license showing the holder's "claims" (Patrimonio Designs LTD/`Shutterstock.com`)*

For example, your license shows your claims of name (identity), address, gender, weight, height, eye color, and more. And again, because your license is created and signed by your state of residence, anyone reading it will trust the claims to be valid.

In short, a token is just a pile of claims that happens to be signed by a trusted issuer, thus removing the need for your service to store user names and passwords. This is huge, of course, as not having to store passwords greatly reduces the risk of your users' passwords being exposed.

The JSON Web Token

There are various formats of software security tokens available, and the current prevailing format is the JSON Web Token (*JWT*, for short). This format contains some header information, a signature, and the token's claims. In plain text, the token's claims might look like this:

```
{
    "Givenname":"Boss",
    "Surname":"Hogg",
    "UserId":"bhogg",
    "Email":"bhogg@example.com",
    "Height":"180cm",
    "Role":"Manager"
}
```

The token's header is used to specify some other things like signature algorithm, expiration date, name of the issue, and a few other attributes.

TOKENS WITH OAUTH2 AND OPENID CONNECT

In this book, we aren't going to cover the implementation details of OAuth integration with identity providers such as Facebook, Twitter, and Google. However, the OAuth2 and OpenId Connect specifications utilize the JWT token format. So as long as you're accepting JWTs, your service will easily accept access tokens generated and signed by these services. In fact, accepting these tokens is really all you need to do in order to support integration with such OAuth providers. Of course, we're skipping over some details, but that's essentially it. Most of the code required to support OAuth is actually found in the client or calling application, not the service.

Since the information in the token is really all we need in order to authorize the user (bhogg is a Manager, for example), our main job here is to validate the token's signature and convert the JWT to a .NET `IPrincipal` object. As you recall from the section on HTTP Basic authentication, we can utilize a message handler to perform this work. Then, once the `HttpContext`'s user is set, the `Authorize` attribute already on our controllers and methods will protect us.

For this particular message handler, we're going to leverage an existing NuGet package instead of implementing it ourselves. You can read about the package here: `https://www.nuget.org/packages/JwtAuthForWebAPI`. Let's dive into the code that leverages this package for our task-management service!

Configuring the JwtAuthForWebAPI Package

With the solution open, run the following command in the Package Manager console:

```
install-package JwtAuthForWebAPI -Pre WebApi2Book.Web.Api
```

This will install our JWT-based NuGet package into the web project. Note that we're using a pre-release version of the package. This is needed because at the time of this writing the underlying JWT library from Microsoft is also still in pre-release. By the time you read this, though, you may need to remove the "-Pre" argument. The `JwtAuthForWebAPI` page on `nuget.org` will indicate the appropriate command to execute.

Once the package is installed, add the following to the `configSections` element in the `web.config` file:

```
<section name="JwtAuthForWebAPI" type="JwtAuthForWebAPI.JwtAuthForWebApiConfigurationSection"/>
```

Then add the following section, below the `configSections` element:

```
<JwtAuthForWebAPI
    AllowedAudience="http://www.example.com"
    Issuer="corp"
    SymmetricKey="cXdlcnR5dWlvcGFzZGZnaGprbHp4Y3ZibmOxMjMONTY=" />
```

In a minute, we will add the code that reads these configuration values. But first, a few words about the `SymmetricKey` property.

You can sign JWTs by using either a private key (of a public/private key pair) or a shared symmetric key. An example of using a public/private key pair is utilizing an X.509 certificate. Certificates have been around for a number of years, and mechanisms are in place for ensuring their authenticity and for storing and managing them securely. In the configuration code just shown, if we wanted to use a certificate instead of a symmetric key, we would use the `SubjectCertificateName` property instead, like this (replacing the `SymmetricKey` property):

```
SubjectCertificateName="CN=JwtAuthForWebAPI Example"
```

One could argue that using certificates for signing these tokens is more secure. Regardless, using a shared symmetric key is easier to demonstrate, so that's what we're doing here. You can create your own symmetric key by running the following statements within PowerShell:

```
$keyBytes=[System.Text.Encoding]::UTF8.GetBytes("qwertyuiopasdfghjklzxcvbnm123456")
[System.Convert]::ToBase64String($keyBytes)
```

The first line creates a byte array from a 32-character string. The second line then converts the array to a base64 encoded string. This method of signing is called "shared key" because, unlike public/private key pairs, the issuer and the service must both have a copy of the same key (i.e., the base64 string).

You are free to use the symmetric key shown earlier, or you can generate your own. You should definitely generate your own if you plan to use a symmetric key in a production environment, though!

Next we need to add the code that configures the associated message handler. Add the following to the bottom of the `RegisterHandlers` method in the `WebApiApplication` class:

```
var builder = new SecurityTokenBuilder();
var reader = new ConfigurationReader();
GlobalConfiguration.Configuration.MessageHandlers.Add(
    new JwtAuthenticationMessageHandler
    {
        AllowedAudience = reader.AllowedAudience,
        Issuer = reader.Issuer,
        SigningToken = builder.CreateFromKey(reader.SymmetricKey)
    });
```

■ **Note** You'll also need to add a `using` directive for `JwtAuthForWebAPI` to satisfy the compiler.

The `SecurityTokenBuilder` object is used to create a `SecurityToken` from a symmetric key. *Note that you can use the same builder to create a* `SecurityToken` *from a certificate subject name.* And the `ConfigurationReader` object is used to read the configuration values we added to the `web.config` file.

And that's it! The JWT-based handler is now configured and added to the ASP.NET Web API message handlers collection. At this point, your service can accept JWTs that are signed with the specific symmetric key, intended for the specified audience, and issued by the specific issuer. The rest of the security-related code we've already added (i.e., the `Authorize` attribute and `IUserSession`) will work exactly as they did before. And because we've added to the handlers collection, our service can now accept either HTTP Basic authentication or JWTs.

Let's make a few JWT-based calls to our service using Fiddler.

Getting a Task Using a JWT

Before we can make a call with Fiddler, we need to first create a valid JWT. If you look in the code associated with this book, you will find a small Console application called `CreateJwt`. The code is very simple, the guts of which look like this:

```
using System;
using System.Diagnostics;
using System.IdentityModel.Protocols.WSTrust;
using System.IdentityModel.Tokens;
using System.Security.Claims;

namespace CreateJwt
{
    internal class Program
    {
        private const string SymmetricKey = "cXdlcnR5dWlvcGFzZGZnaGprbHp4Y3ZibmOxMjMONTY=";

        private static void Main(string[] args)
        {
            var key = Convert.FromBase64String(SymmetricKey);
            var credentials = new SigningCredentials(
                new InMemorySymmetricSecurityKey(key),
                "http://www.w3.org/2001/04/xmldsig-more#hmac-sha256",
                "http://www.w3.org/2001/04/xmlenc#sha256");

            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(new[]
                {
                    new Claim(ClaimTypes.Name, "bhogg"),
                    new Claim(ClaimTypes.GivenName, "Boss"),
                    new Claim(ClaimTypes.Surname, "Hogg"),
                    new Claim(ClaimTypes.Role, "Manager"),
                    new Claim(ClaimTypes.Role, "SeniorWorker"),
                    new Claim(ClaimTypes.Role, "JuniorWorker")
                }),
                TokenIssuerName = "corp",
                AppliesToAddress = "http://www.example.com",
                SigningCredentials = credentials,
                Lifetime = new Lifetime(DateTime.UtcNow, DateTime.UtcNow.AddYears(10))
            };

            var tokenHandler = new JwtSecurityTokenHandler();
            var token = tokenHandler.CreateToken(tokenDescriptor);
            var tokenString = tokenHandler.WriteToken(token);

            Console.WriteLine(tokenString);
            Debug.WriteLine(tokenString);

            Console.ReadLine();
        }
    }
}
```

Note first that the symmetric key matches that which we placed in the `web.config` file in the previous section. Second, we are specifying a few claims—the roles being the most relevant for our discussion at this point. And third, we are setting the expiration date to 10 years from the current date. Obviously, a production-ready lifetime might be a little shorter!

And finally, the console app writes the generated JWT to the console (and debug window). The token generated when writing this book was

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJjb3JwIiwiYXVkIjoiaHR0cDovL3d3dy
5leGFtcGxlLmNvbSIsIm5iZiI6MTQwMDU1Mzc1NywiZXhwIjoxNzE2MTcyOTU3LCJ1bmlxdWVfbmFtZ
SI6ImJob2dnIiwiZ2l2ZW5fbmFtZSI6IkJvc3MiLCJmYW1pbHlfbmFtZSI6IkhvZ2ciLCJyb2xlIjpb
Ik1hbmFnZXIiLCJKdW5pb3JXb3JrZXIiXX0.Ls73kz8OrCaCNqzc3K32BVO9_LnJDL8c1g5AXKIzn8w
```

We will need to alter our API call by changing the Authorization header a bit. First, change the authorization scheme to Bearer. And then replace the base64-encoded user name and password with the generated JWT string. The resulting request looks like this:

```
GET http://localhost:52975/api/v1/tasks/2 HTTP/1.1
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJjb3JwIiwiYXVkIjoiaHR0cD
ovL3d3dy5leGFtcGxlLmNvbSIsIm5iZiI6MTQwMDU1Mzc1NywiZXhwIjoxNzE2MTcyOTU3LCJ1bmlxdWVfbmFtZSI6Im
Job2dnIiwiZ2l2ZW5fbmFtZSI6IkJvc3MiLCJmYW1pbHlfbmFtZSI6IkhvZ2ciLCJyb2xlIjpbIk1hbmFnZXIiLCJKdW5pb3JX
b3JrZXIiXX0.Ls73kz8OrCaCNqzc3K32BVO9_LnJDL8c1g5AXKIzn8w
```

■ **Note** The formatting here places the JWT on a separate line, but in reality there is only a space between "Bearer" and the JWT value, not a new line.

You can experiment with the token generation code in the `CreateJwt` console application, removing a role or two. In doing so, you will get the same unauthorized responses that we showed earlier in this chapter.

SSL, XSS, CORS, and CSRF

We would be remiss if we didn't at least acknowledge some of the security concerns within web applications today. Vulnerabilities dealing with cross-site scripting and cross-site request forgery, and supporting things like cross-origin requests and SSL, are most certainly at the top of web site architecture concerns. But given that we're talking about a REST service in this book, these concerns take on a slightly different meaning. In truth, most of them are more related to HTTP requests coming from web pages (i.e., web forms and AJAX calls) than they are API calls originating from non-web clients.

As such, we will cover topics like these in Chapter 9, when we demonstrate the consumption of our new Task Management service by a couple different web pages.

Summary

In this chapter, you learned how to leverage the power of ASP.NET Web API message handlers to implement a service-wide feature: authentication. You built a handler for supporting HTTP Basic authentication by hand. And then, with just a few lines of code, you configured a NuGet package to provide support for JSON Web Token based security. Then you utilized the `Authorize` attribute at both the controller and controller action level to protect your service.

You also learned how to implement auditing using the new ASP.NET Web API async filter feature, as well as global handling of our own custom exceptions.

In the next chapter, we will explore how to support some miscellaneous service features like partial resource updates, resource relationships, and a few others.