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 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 two 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 scenarios we will cover in this chapter:

Table 6-1. Scenarios Used to Illustrate Security

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

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

* User: The end user of the task-management service. 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 credentials. In order to do so, the caller must provide two basic pieces of information: who the user claims to be, and how that claim can be verified.

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 username and password. Those values are essentially plaintext, using only base64 encoding for simple obfuscation.  This generally requires SSL transport security (i.e., an endpoint that exposes an HTTPS address) to protect 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 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 when third-party token issuers are involved (e.g., OpenID). This relieves your service of the burden of verifying a user’s credentials.  Here's how it works:  The caller first verifies their username and password using a token issuer that your service trusts. Upon successful verification, the 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, 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 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 only have to deal 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, configuring SSL transport security is outside the scope of this book, because we are striving to keep the focus on ASP.NET Web API.

Finally, we will also support a form of token-based security in the task-management service, as 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 OAth). 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 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).

Figure 6-3. An Example User’s Claims

|  |  |
| --- | --- |
| Claim type | Example claim value |
| Email | [bob@gmail.com](mailto: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 | SeniorWorker |

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 isssuer 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 thread that contains the current user’s identity and associated claims (e.g., userId, email, firstname, lastname, and roles). Each web request executes on its own thread, so each request will execute within the context of the caller’s principal.
4. All "downstream" code checks the current thread’s principal to determine if/how processing is allowed to continue. If processing is not allowed to continue, then 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!

# 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 AddTask by adding an Authorize attribute (i.e., an "authorization filter") to it as follows. 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 by this:

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 attribute on the AddTask method. This causes it to ensure that a security principal containing a Manager role has been established on the thread 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 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 thread. 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 thread. Remember, the principal must be established on the current thread 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 that it delegates some of its principal-building responsibilities to. 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.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>(); }

}

public bool SetPrincipal(string username, string password)

{

var principal = GetPrincipal(username);

if (principal == 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(string username)

{

var identity = new GenericIdentity(username, Constants.SchemeTypes.Basic);

username = 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);

}

}

}

Then wire this up so 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();

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 soon, 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 above.

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 3rd 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 - accessable 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 a single user credential, username; 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). This is a book about ASP.NET Web API, not credential management and verification; therefore, we will move on to our custom handler. Go ahead and implement it 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 credentials = GetCredentials(authHeader);

if (credentials.Length != ExpectedCredentialCount)

{

return false;

}

return \_basicSecurityService.SetPrincipal(credentials[UsernameIndex], credentials[PasswordIndex]);

}

public string[] GetCredentials(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.

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.StatusCode == HttpStatusCode.Unauthorized ? CreateUnauthorizedResponse() : response;

The first line passes processing along to the next handler and waits for the response. The second line analyzes the response and modifies it if the response indicates that the request was unauthorized (e.g., if the user is not in a Manager role in our Create a Task scenario). It modifies the response by using the CreateUnauthorizedResponse method.

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.

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 off to the security service we implemented earlier to do the actual work of setting the principal.
* 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 add it 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>()));

}

}

}

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 caller’s credentials and setup a corresponding principal using a trivial implementation in the BasicSecurityService. Now let's 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 JuniorWorker role. Note, "amRvZTppZ25vcmVk" represents the credentials for "jdoe", the junior worker:

POST Request - JuniorWorker (abbreviated)

POST http://localhost:61589/api/v1/tasks HTTP/1.1

Authorization: Basic amRvZTppZ25vcmVk

Content-Type: text/json

{"Subject":"Fix something important"}

POST Response - JuniorWorker (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 two remaining scenarios from Table 6-1 leveraging the security infrastructure we've put in place. After that, we'll show how we can add on 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.

# 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:

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 |

With that as an introduction, we're ready to begin implementation, so let's by start adding support to activate a task. We'll follow our usual approach of adding dependencies first, and the first dependency we'll add is the query processor:

ITaskByIdQueryProcessor Interface

using WebApi2Book.Data.Entities;

namespace WebApi2Book.Data.SqlServer.QueryProcessors

{

public interface ITaskByIdQueryProcessor

{

Task GetTask(long taskId);

}

}

TaskByIdQueryProcessor Class

using NHibernate;

using WebApi2Book.Data.Entities;

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 bottom of 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 so trivial that it does not merit further discussion in and of itself. However, astute readers familiar with the Strategy Pattern may notice that the query processor interfaces should actually be located in the WebApi2Book.Data project, not in the WebApi2Book.Data.SqlServer project. Yes, we took this little shortcut for the sake of making it easier to follow along with the implementation (i.e., not as much switching between projects).

The next dependency we'll add implements all of the "business logic" required to activate a task. Implement as follows:

IStartTaskWorkflowProcessor Interface

using WebApi2Book.Web.Api.Models;

namespace WebApi2Book.Web.Api.MaintenanceProcessing

{

public interface IStartTaskWorkflowProcessor

{

Task StartTask(long taskId);

}

}

StartTaskWorkflowProcessor Class

# 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.

# Dumping ground of raw notes

Some things we'll be illustrating by way of the scenarios:

\*Basic Auth

\*Bearer Auth

\*Authorize attribute

\*CORS (we could wait until Ch9 for this, when we hook into the UI)

\*ASP.NET Identity (optional)?

\*Certificates, which are required for SSL (optional)?

\*Require SSL attribute (optional)

\* CSRF??? What are you planning to do about this? Is this just a sidebar somewhere

Finally, we'll implement a task GET to demonstrate removing sensitive data from response. User must have a SeniorWorker role to see who is assigned to a task.

Scenarios used to illustrate security features:

\* Only managers can create and delete tasks.

\* Non-resource API using REST. We will audit task reactivations using an async filter. Also, user must have a SeniorWorker role to start, complete, or reactivate task.

\* Removing sensitive data from response. User must have a SeniorWorker role to see who is assigned to a task.