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.

# 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 caller, 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 throw a security exception. The security exception will be communicated 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

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…

## The Message Handler

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 (as we saw earlier).

Before we implement our message handler, we need to add the security service that it delegates some of its principal-building responsibilities to (add the following types):

ISecurityService Interface

namespace WebApi2Book.Web.Api.Security

{

public interface ISecurityService

{

bool SetPrincipal(string username, string password);

}

}

SecurityService 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 SecurityService : ISecurityService

{

private readonly ILog \_log;

public SecurityService(ILogManager logManager)

{

\_log = logManager.GetLog(typeof (SecurityService));

}

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);

var principal = GetPrincipal(username);

if (user == null || 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);

}

public virtual User GetUser(string username)

{

username = username.ToLowerInvariant();

return

Session.QueryOver<User>().Where(x => x.Username == username).SingleOrDefault();

}

}

}

TODO: Describe the security service

And now implement the handler 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 ISecurityService \_securityService;

public BasicAuthenticationMessageHandler(ILogManager logManager, ISecurityService securityService)

{

\_securityService = securityService;

\_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 \_securityService.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;

}

}

}

Todo…

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

public class BasicAuthenticationMessageHandler : DelegatingHandler

{

public const string BasicScheme = "Basic";

public const string ChallengeAuthenticationHeaderName = "WWW-Authenticate";

public const char AuthorizationHeaderSeparator = ':';

private readonly IMembershipInfoProvider\_membershipAdapter;

private readonly ISessionFactory \_sessionFactory;

public BasicAuthenticationMessageHandler(IMembershipInfoProvidermembershipAdapter, ISessionFactory sessionFactory)

{

\_membershipAdapter = membershipAdapter;

\_sessionFactory = sessionFactory;

}

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

{ ….

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

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

var authHeader = request.Headers.Authorization;

if (authHeader == null)

{

return CreateUnauthorizedResponse();

}

if (authHeader.Scheme != BasicScheme)

{

return CreateUnauthorizedResponse();

}

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

private Task<HttpResponseMessage> CreateUnauthorizedResponse()

{

var response = new HttpResponseMessage(HttpStatusCode.Unauthorized);

response.Headers.Add(ChallengeAuthenticationHeaderName, BasicScheme);

var taskCompletionSource = new TaskCompletionSource<HttpResponseMessage>();

taskCompletionSource.SetResult(response);

return taskCompletionSource.Task;

}

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

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

var encodedCredentials = authHeader.Parameter;

var credentialBytes = Convert.FromBase64String(encodedCredentials);

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

var credentialParts = credentials.Split(AuthorizationHeaderSeparator);

if(credentialParts.Length != 2)

{

return CreateUnauthorizedResponse();

}

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

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

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

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

{

return CreateUnauthorizedResponse();

}

SetPrincipal(username);

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

private void SetPrincipal(string username)

{

var roles = \_membershipAdapter.GetRolesForUser(username);

var user = \_membershipAdapter.GetUser(username);

User modelUser;

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

{

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

}

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

var principal = new GenericPrincipal(identity, roles);

Thread.CurrentPrincipal = principal;

if (HttpContext.Current != null)

{

HttpContext.Current.User = principal;

}

}

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

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

private GenericIdentity CreateIdentity(string username, User modelUser)

{

var identity = new GenericIdentity(username, BasicScheme);

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

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

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

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

return identity;

}

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

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

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

[Authorize(Roles = "Administrators")]

public HttpResponseMessage Post(HttpRequestMessage request, Category category)

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

private static void RegisterServices(IKernel kernel)

{

var containerConfigurator = new NinjectConfigurator();

containerConfigurator.Configure(kernel);

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

}

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

# IUserSession

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

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

public interface IUserSession

{

Guid UserId { get; }

string Firstname { get; }

string Lastname { get; }

string Username { get; }

string Email { get; }

}

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

public class UserSession : IUserSession

{

public UserSession(ClaimsPrincipal principal)

{

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

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

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

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

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

}

public Guid UserId { get; private set; }

public string Firstname { get; private set; }

public string Lastname { get; private set; }

public string Username { get; private set; }

public string Email { get; private set; }

}

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

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

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

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

Next, add this corresponding CreateUserSession()method:

private IUserSession CreateUserSession(IContext arg)

{

return new UserSession(Thread.CurrentPrincipal as GenericPrincipal);

}

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

private readonly ICommonRepository \_commonRepository;

private readonly IHttpTaskFetcher \_taskFetcher;

private readonly IUserSession \_userSession;

public TasksController(

ICommonRepository commonRepository,

IHttpTaskFetcher taskFetcher,

IUserSession userSession)

{

\_commonRepository = commonRepository;

\_taskFetcher = taskFetcher;

\_userSession = userSession;

}

# Summary

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

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

# 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

The token handler can be used to demonstrate securing non-resource API using REST. That will be under another heading.

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.