CHAPTER 12

Authenticating API Clients

In this chapter, I explain how ASP.NET Core Identity can be used to authenticate API clients using a simple JavaScript application. Some projects can rely on standard ASP.NET Core user authentication for their clients, and I demonstrate how this works. For other projects, I explain how to provide authentication using cookies and using bearer tokens, which are useful for clients that don't support cookies. Table 12-1 puts API authentication in context.

Table 12-1. *Putting API Authentication in Context*

Question	Answer
What is it?	API authentication restricts access to the controllers that provide direct access to data, rather than via HTML content. These controllers typically support RESTful web services.
Why is it useful?	Many web services provide data or support operations that should not be publicly accessible. Restricting access by requiring API requests to be authenticated allows an effective authorization policy to be defined.
How is it used?	Access to the API controller is restricted using the `Authorize` attribute. How requests are authenticated varies based on the clients that the applications support.
Are there any pitfalls or limitations?	There are no well-defined standards to describe how web services should be designed or authorized, which can be a problem when you need to support a range of third-party clients.
Are there any alternatives?	Not all applications support APIs, but when they are supported, some kind of authentication is required.

Table 12-2 summarizes the chapter.

Table 12-2. *Chapter Summary*

Problem	Solution	Listing
Authenticate JavaScript API clients that are delivered by the ASP.NET Core server	Rely on the standard authentication cookie that is created when the user signs into the application.	1–8
Return status code responses to API clients	Define a handler that overrides the default behavior for challenge and forbidden responses.	9–13
Authenticate API clients with a cookie	Create an authentication controller and use the sign-in manager's methods to sign the user into the application and return a JSON response. Ensure the JavaScript client includes the cookie in subsequent requests.	14–19
Authenticate API clients without a cookie	Use a bearer token, which the client can present in subsequent requests.	20–25

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_12

Preparing for This Chapter

This chapter uses the IdentityApp project from Chapter 11. To demonstrate API client authentication, I need to set up an API controller. Add a class file named `ValuesController.cs` to the `Controllers` folder with the code shown in Listing 12-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/Apress/pro-asp.net-core-identity`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 12-1. The ValuesController.cs File in the Controllers Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;

namespace IdentityApp.Controllers {

    [ApiController]
    [Route("/api/data")]
    public class ValuesController : ControllerBase {
        private ProductDbContext DbContext;

        public ValuesController(ProductDbContext dbContext) {
            DbContext = dbContext;
        }

        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() => DbContext.Products;

        [HttpPost]
        public async Task<IActionResult> CreateProduct([FromBody]
                ProductBindingTarget target) {
            if (ModelState.IsValid) {
                Product product = new Product {
                    Name = target.Name, Price = target.Price,
                        Category = target.Category
                };
                await DbContext.AddAsync(product);
                await DbContext.SaveChangesAsync();
                return Ok(product);
            }
            return BadRequest(ModelState);
        }

        [HttpDelete("{id}")]
```

```
        public Task DeleteProduct(long id) {
            DbContext.Products.Remove(new Product { Id = id });
            return DbContext.SaveChangesAsync();
        }
    }

    public class ProductBindingTarget {
        [Required]
        public string Name { get; set; }

        [Required]
        public decimal Price { get; set; }

        [Required]
        public string Category { get; set; }
    }
}
```

This is a basic API controller that provides access to the data in the product database, which has only been used to generate content in MVC views and Razor Pages until now. I don't need a full set of API functions for this chapter because the focus is on authentication. There are actions to retrieve all the products in the database, delete a product from the database, and add a product to the database. The `ProductBindingTarget` class is used to ensure that only selected properties defined by the `Product` class will be used by the model binding process.

Creating the JavaScript API Client

For this chapter, I use a pure JavaScript client, which will allow me to focus on the authentication process without the distraction of a large framework getting in the way. Create the `wwwroot/js` folder and add to it a JavaScript file named `network.js` with the content shown in Listing 12-2.

■ **Note** I describe how to handle bearer tokens in Angular and React in my books for those frameworks, *Pro Angular* and *Pro React*.

Listing 12-2. The Contents of the network.js File in the wwwroot/js Folder

```
const baseUrl = "https://localhost:44350/api/data";

export const loadData = async function (callback, errorHandler) {
    const response = await fetch(baseUrl, {
        redirect: "manual"
    });
    processResponse(response, async () => callback(await response.json()),
        errorHandler);
}

export const createProduct = async function (product, callback, errorHandler) {
    const response = await fetch(baseUrl, {
```

```
        method: "POST",
        body: JSON.stringify(product),
        headers: {
            "Content-Type": "application/json"
        }
    });
    processResponse(response, callback, errorHandler);
}

export const deleteProduct = async function (id, callback, errorHandler) {
    const response = await fetch(`${baseUrl}/${id}`, {
        method: "DELETE"
    });
    processResponse(response, callback, errorHandler);
}

function processResponse(response, callback, errorHandler) {
    if (response.ok) {
        callback();
    } else {
        errorHandler(response.status);
    }
}
```

The functions defined in this file are responsible for sending HTTP requests to the API controller. I have used the Fetch API, which is supported by modern browsers and provides a more usable alternative to the traditional `XmlHttpRequest` object.

Add a JavaScript file named `client.js` to the `wwwroot/js` folder with the code shown in Listing 12-3.

Listing 12-3. The Contents of the client.js File in the wwwroot/js Folder

```
import * as network from "./network.js";

const columns = ["ID", "Name", "Category", "Price"];
let tableBody;
let errorElem;

HTMLElement.prototype.make = function (...types) {
    return types.reduce((lastElem, elemType) =>
        lastElem.appendChild(document.createElement(elemType)), this);
}

function showError(err) {
    errorElem.innerText = `Error: ${err}`;
    errorElem.classList.add("m-2", "p-2");
}

function clearError(err) {
    errorElem.innerText = "";
    errorElem.classList.remove("m-2", "p-2");
}
```

```
function createStructure() {
    const targetElement = document.getElementById("target");
    targetElement.innerHTML = "";
    errorElem = targetElement.make("div");
    errorElem.classList.add("h6", "bg-danger", "text-center", "text-white");
    return targetElement;
}

function createContent() {
    const targetElement = createStructure();
    const table = targetElement.make("table");
    table.classList.add("table", "table-sm", "table-striped", "table-bordered");
    const headerRow = table.make("thead", "tr");
    columns.concat([""]).forEach(col => {
        const th = headerRow.make("th");
        th.innerText = col;
    });
    tableBody = table.make("tbody");
    const footerRow = table.make("tfoot", "tr");
    footerRow.make("td");
    columns.filter(col => col != "ID").forEach(col => {
        const input = footerRow.make("td", "input");
        input.name = input.id = col;
        input.placeholder = `Enter ${col.toLowerCase()}`;
    });
    const button = footerRow.make("td", "button");
    button.classList = "btn btn-sm btn-success";
    button.innerText = "Add";
    button.addEventListener("click", async () => {
        const product = {};
        columns.forEach(col => product[col] = document.getElementById(col)?.value);
        await network.createProduct(product, populateTable, showError);
    });
}

function createTableContents(products) {
    tableBody.innerHTML = "";
    products.forEach(p => {
        const row = tableBody.appendChild(document.createElement("tr"));
        columns.forEach(col => {
            const cell = row.appendChild(document.createElement("td"));
            cell.innerText = p[col.toLowerCase()];
        });
        const button = row.appendChild(document.createElement("td")
            .appendChild(document.createElement("button")));
        button.classList.add("btn", "btn-sm", "btn-danger");
        button.textContent = "Delete";
        button.addEventListener("click", async () =>
            await network.deleteProduct(p.id, populateTable, showError));
    });
}
```

```
async function populateTable(products) {
    clearError();
    await network.loadData(createTableContents, showError);
}

document.addEventListener("DOMContentLoaded", () => {
    createContent();
    populateTable();
})
```

Usually, I write code so that it is easy to understand, but that is not the case for this JavaScript code, which uses the browser's Domain Object Model (DOM) API to create a simple table, which is populated with data from the API controller. This is not a book about JavaScript, and I don't describe how it works because the parts that are important for this chapter are all contained in the `network.js` file.

To deliver the JavaScript code to the browser, add a Razor Page named `JSClient.cshtml` to the `Pages` folder with the content shown in Listing 12-4.

Listing 12-4. The Contents of the JSClient.cshtml File in the Pages Folder

```
@page

<div id="target" class="m-3">
    Loading JavaScript client...
</div>
<script type="module" src="/js/client.js"></script>
```

The `div` element will be used to display the content generated by the JavaScript code, which is specified by the `script` element.

Open a new PowerShell command prompt and run the commands shown in Listing 12-5 to reset the application and Identity databases.

Listing 12-5. Resetting the Databases

```
dotnet ef database drop --force --context ProductDbContext
dotnet ef database drop --force --context IdentityDbContext
dotnet ef database update --context ProductDbContext
dotnet ef database update --context IdentityDbContext
```

Use the PowerShell prompt to run the command shown in Listing 12-6 in the `IdentityApp` folder to start the application.

Listing 12-6. Running the Example Application

```
dotnet run
```

Open a web browser, request `https://localhost:44350/Identity/Admin`, and sign in as `admin@example.com` using `mysecret` as the password. When you sign in, you will be redirected o the administration dashboard. Click the Seed Database, which will update the dashboard to indicate there are four users in the store, as shown in Figure 12-1.

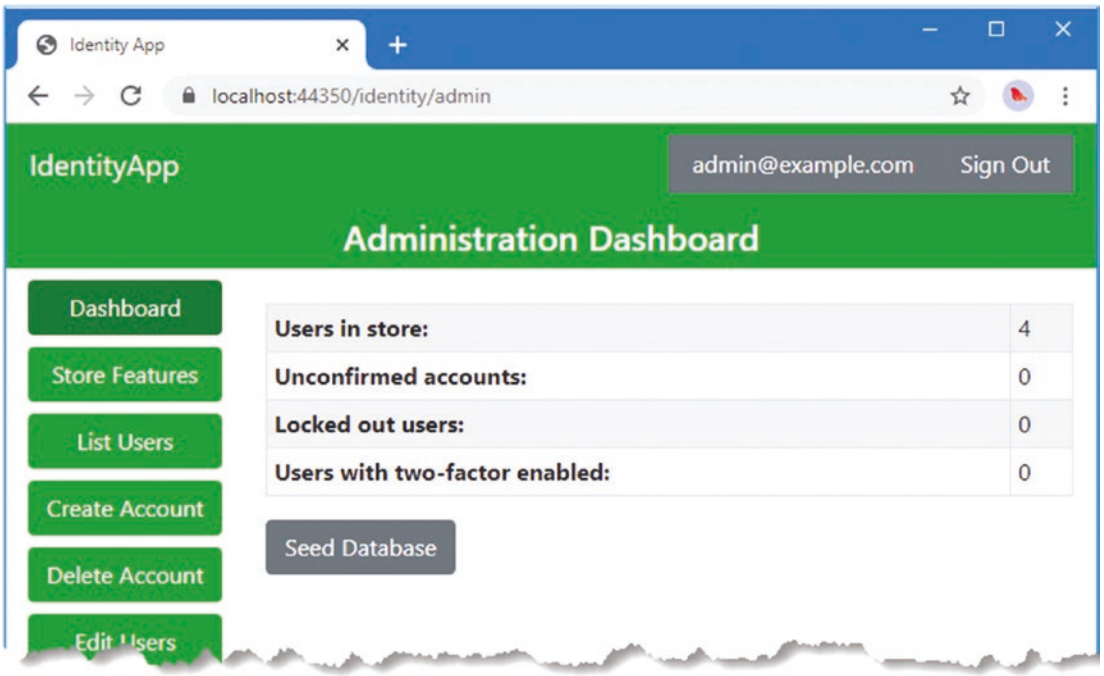

Figure 12-1. *Running the example application*

Request `https://localhost:44350/jsclient` and you will see the output from the JavaScript client, as shown in Figure 12-2.

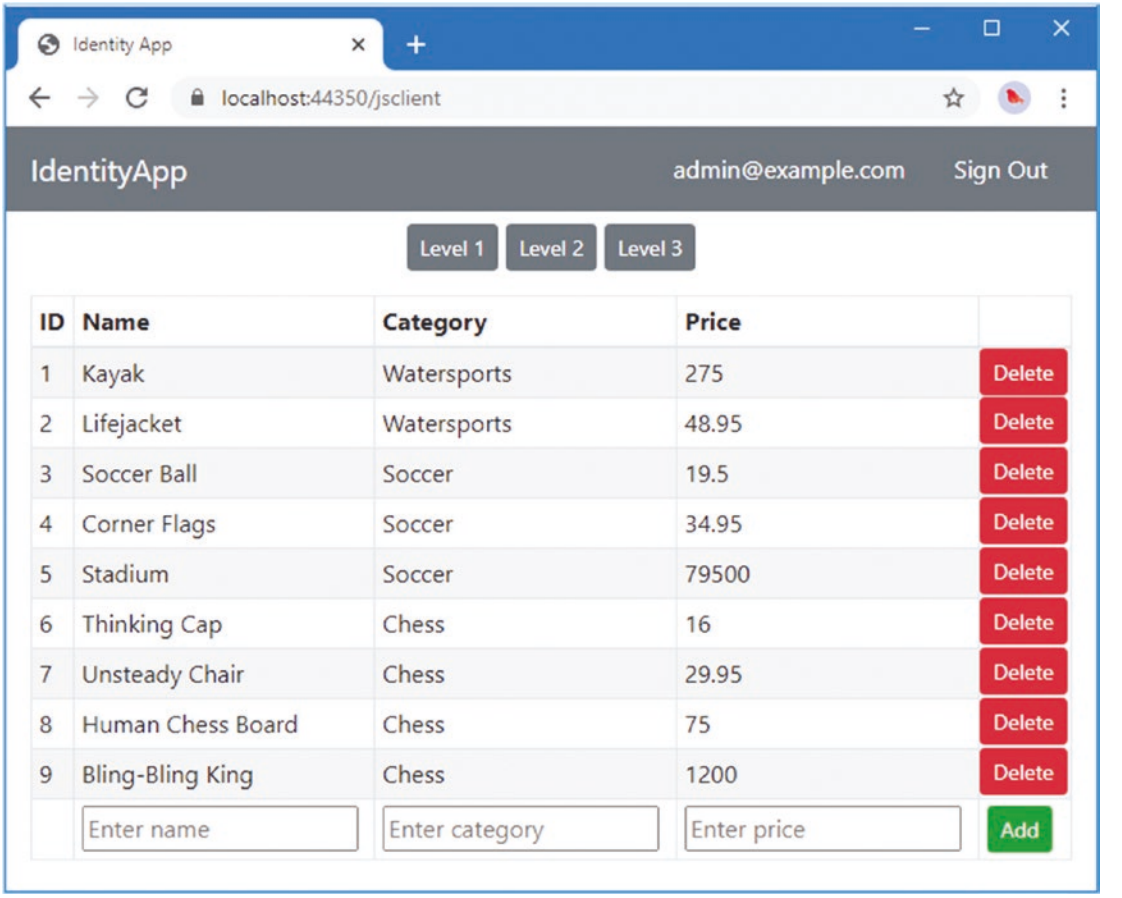

Figure 12-2. *The JavaScript API client*

Using Simple Authentication for JavaScript Clients

The easiest way to restrict access to API controllers is to rely on the standard ASP.NET Core and Identity features used in earlier chapters. This approach has some drawbacks, which I explain shortly, but it has the advantage of allowing JavaScript clients to benefit from authentication and authorization without needing to handle any of the details directly.

To demonstrate, Listing 12-7 decorates the API controller with the `Authorize` attribute, which will allow only signed-in users to access the actions defined by the web service controller.

Listing 12-7. Restricting Access in the ValuesController.cs File in the Controllers Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
```

```
namespace IdentityApp.Controllers {

    [Authorize]
    [ApiController]
    [Route("/api/data")]
    public class ValuesController : ControllerBase {
        private ProductDbContext DbContext;

        // ...methods omitted for brevity...
    }

    public class ProductBindingTarget {
        [Required]
        public string Name { get; set; }

        [Required]
        public decimal Price { get; set; }

        [Required]
        public string Category { get; set; }
    }
}
```

This technique requires the same restriction to be applied to the Razor Page or action method that contains the `script` element for the JavaScript code, as shown in Listing 12-8.

Listing 12-8. Restricting Access in the JSClient.cshtml File in the Pages Folder

```
@page
@model IdentityApp.Pages.JSClientModel
@attribute [Microsoft.AspNetCore.Authorization.Authorize]

<div id="target" class="m-3">
    Loading JavaScript client...
</div>

<script type="module" src="/js/client.js"></script>
```

Restart ASP.NET Core and request `https://localhost:44350/jsclient`. Sign in as `alice@example.com` with the password `mysecret` and you will be redirected to the page that contains the JavaScript code, which will request the data from the API controller, as shown in Figure 12-3.

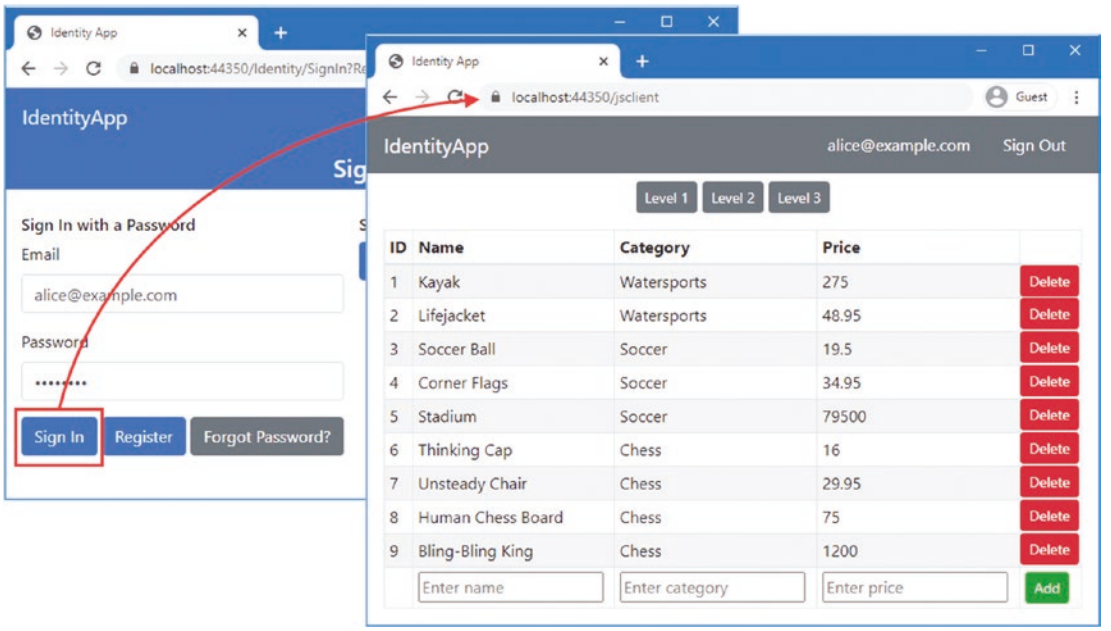

Figure 12-3. *Using the standard authentication cookie for API authorization*

When the user signs in, ASP.NET Core adds a cookie to the response that is included in future requests, including those made by the JavaScript code. ASP.NET Core doesn't differentiate between requests initiated directly by the user and requests initiated by JavaScript code and authenticates and authorizes them in the same way.

This approach works with almost no effort, but it can only be used for JavaScript applications that are delivered using ASP.NET Core and expect authentication to be handled for them. This approach doesn't work for JavaScript clients that expect to handle authentication directly, because the only way to sign in to the application is by submitting a form that is rendered by a Razor Page. You could write a JavaScript client that submits form data, but it is easier to create an authentication API as I demonstrate later in this chapter.

Using standard authentication and authorization for API clients isn't as terrible as it may seem. It is suitable for projects where the JavaScript and ASP.NET Core parts of the application are developed side by side and you know that the JavaScript client will always be able to rely on an ASP.NET Core cookie. Despite its limitations, this scenario covers a lot of projects, and you shouldn't dismiss it out of hand. Any situation in which you can deliver your project requirements without any additional effort counts as a win.

Returning Status Code Responses for API Clients

API clients typically expect web services to follow the broad principles of Representational State Transfer (REST) and use HTTP status codes to indicate the outcome of operations. That isn't what ASP.NET Core does by default because it wants to present meaningful HTML content to the user, even—or especially—when there is a problem.

To see the problem this causes, open a PowerShell command prompt, and run the command shown in Listing 12-9 to send an HTTP request to the web service.

Listing 12-9. Sending an HTTP request

```
Invoke-WebRequest -Uri https://localhost:44350/api/data
```

This command sends an HTTP GET request to the API controller, using the same URL as the JavaScript client. The request sent by the command doesn't include an authentication cookie and so ASP.NET Core generates a challenge response. However, because ASP.NET Core is expecting to present content to a user, the challenge response is intercepted and replaced with a redirection that instructs the browser to request the sign-in page, which can be seen in the output from the command.

```
...
StatusCode        : 200
StatusDescription : OK
Content           :
                    <!DOCTYPE html>
                    <html>
                    <head>
                        <meta name="viewport" content="width=device-width" />
                        <title>Identity App</title>
                        <link href="/lib/twitter-bootstrap/css/bootstrap.min.css"
...
```

The results are inconsistent and depend on the client that made the request. Some clients follow the redirection and end up with a 200 OK response that contains the HTML for the sign-in pace—this is what happens with the command in Listing 12-9. Clients that expect an HTTP status code response will believe the request succeeded because they received a 200 status code. This type of client will often attempt to parse the HTML response as JSON data and encounter an error.

Other clients won't follow the redirection but are equally stuck because they have no way to get the data they require, and the redirection response doesn't provide any information about what to do to resolve the issue.

To make matters worse, some requests will receive a status code response. Use the PowerShell command prompt to run the command shown in Listing 12-10. Make sure the entire command is on a single line.

Listing 12-10. Sending an HTTP Request with a Header

```
Invoke-WebRequest -Uri https://localhost:44350/api/data -Headers @{"X-Requested-
With"="XMLHttpRequest"}
```

This command sends a GET request to the same URL but with the addition of a header named `X-Requested-With` with a value of `XMLHttpRequest`. The response from ASP.NET Core is a 401 status code response, like this:

```
...
Invoke-WebRequest : The remote server returned an error: (401) Unauthorized.
...
```

The `XMLHttpRequest` header value refers to the JavaScript object with the same name, which is the traditional way to make HTTP requests in client-side applications. Some browsers and JavaScript frameworks set the `X-Requested-With` header to `XMLHttpRequest` to help servers identify HTTP requests

made with JavaScript code. Unfortunately, you can't rely on this header being set because the Fetch API has been introduced as a modern alternative to the `XmlHttpRequest` object (Fetch is the API I used in the example JavaScript client) and because not all browsers and frameworks set the header even when `XmlHttpRequest` is being used.

To change how API clients are handled by the example application, add the statement shown in Listing 12-11 to the `Startup` class. (The method called by this statement has not yet been defined, so you will see an error in the code editor.)

Listing 12-11. Handling API Clients in the Startup.cs File in the IdentityApp Folder

```
...
services.ConfigureApplicationCookie(opts => {
    opts.LoginPath = "/Identity/SignIn";
    opts.LogoutPath = "/Identity/SignOut";
    opts.AccessDeniedPath = "/Identity/Forbidden";
    opts.Events.DisableRedirectionForApiClients();
});
...
```

The `CookieAuthenticationOptions` class defines an `Events` property, which returns an `CookieAuthenticationEvents` object that allows handlers to be defined for key events during authentication and authorization. There are four properties for dealing with API clients, as described in Table 12-3.

Table 12-3. *The CookieAuthenticationEvents Event Handler Properties for API Clients*

Name	Description
`OnRedirectToLogin`	This property is assigned a handler that is called when the user should be redirected to the sign-in URL. The default handler sends a status code response for requests with the `X-Requested-With` header and performs the redirection for all other requests.
`OnRedirectToAccessDenied`	This property is assigned a handler that is called when the user should be redirected to the access denied URL. The default handler sends a status code response for requests with the `X-Requested-With` header and performs the redirection for all other requests.
`OnRedirectToLogout`	This property is assigned a handler that is called when the user should be redirected to the sign-out URL. The default handler sends a 200 OK response with a `Location` header set to the sign-out URL and performs the redirection for all other requests.
`OnRedirectToReturnUrl`	This property is assigned a handler that is called when the user should be redirected to the URL that triggered a challenge response. The default handler sends a 200 OK response with a `Location` header set to the sign-out URL and performs the redirection for all other requests.

The handler functions assigned to the properties described in Table 12-3 receive an instance of the `RedirectContext<CookieAuthenticationOptions>` class, whose most useful properties are described in Table 12-4.

Table 12-4. *Useful RedirectContext<CookieAuthenticationOptions> Properties*

Name	Description
`HttpContext`	This property returns the `HttpContext` object for the current request.
`Request`	This property returns the `HttpRequest` object that describes the request.
`Response`	This property returns the `HttpRequest` object that describes the response.
`RedirectUri`	This property returns the URL to which the user should be directed.

To define the extension method that I invoked in Listing 12-11, add a class file named `CookieAuthEventsExtensions.cs` to the `IdentityApp` folder and add the code shown in Listing 12-12.

Listing 12-12. The Contents of the CookieAuthEventsExtensions.cs File in the IdentityApp Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace IdentityApp {

    public static class CookieAuthEventsExtensions {

        public static void DisableRedirectionForApiClients(this
                CookieAuthenticationEvents events) {
            events.OnRedirectToLogin = ctx =>
                SelectiveRedirect(ctx, StatusCodes.Status401Unauthorized);
            events.OnRedirectToAccessDenied = ctx =>
                SelectiveRedirect(ctx, StatusCodes.Status403Forbidden);
            events.OnRedirectToLogout = ctx =>
                SelectiveRedirect(ctx, StatusCodes.Status200OK);
            events.OnRedirectToReturnUrl= ctx =>
                SelectiveRedirect(ctx, StatusCodes.Status200OK);
        }

        private static Task SelectiveRedirect(
                RedirectContext<CookieAuthenticationOptions> context, int code) {
            if (IsApiRequest(context.Request)) {
                context.Response.StatusCode = code;
                context.Response.Headers["Location"] = context.RedirectUri;
            } else {
                context.Response.Redirect(context.RedirectUri);
            }
            return Task.CompletedTask;
        }

        private static bool IsApiRequest(HttpRequest request) {
            return request.Path.StartsWithSegments("/api");
        }
    }
}
```

The `DisableRedirectionForApiClients` method assigns new handler functions to the four properties defined in Table 12-3. The outcome of each handler is the same, such that API clients are sent a status code response with a `Location` header, while other clients receive the redirection. The `Location` header provides the API client with the redirection URL without forcing them to request its contents.

The difference is the way that API clients are identified, which is done by the `IsApiRequest` method. Instead of looking for the `X-Requested-With` header, the request path is examined, and if it begins with `/api`, then the client will get status code responses. This is a shift in approach from trying to determine how the request originated to determining what the request is for: if a request is for an `/api` URL, then the client is assumed to want status code responses.

Restart ASP.NET Core and use a PowerShell prompt to run the command shown in Listing 12-13.

Listing 12-13. Sending a Request

```
Invoke-WebRequest -Uri https://localhost:44350/api/data
```

The request won't be authorized because it doesn't contain an ASP.NET Core cookie, but the code in Listing 12-12 detects the request for a URL that starts with `/api` and sends a 401 response.

```
...
Invoke-WebRequest : The remote server returned an error: (401) Unauthorized.
...
```

Authenticating API Clients Directly

Not all API clients can be delivered via an authenticated Razor Page or view and must take responsibility for handling authentication directly.

Preparing ASP.NET Core for Direct API Client Authentication

Add a class file named `ApiAuthController.cs` to the `Controllers` folder and use it to create the controller shown in Listing 12-14.

Listing 12-14. The Contents of the ApiAuthController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using SignInResult = Microsoft.AspNetCore.Identity.SignInResult;

namespace IdentityApp.Controllers {

    [ApiController]
    [Route("/api/auth")]
    public class ApiAuthController: ControllerBase {
        private SignInManager<IdentityUser> SignInManager;

        public ApiAuthController(SignInManager<IdentityUser> signMgr) {
            SignInManager = signMgr;
        }
```

```
        [HttpPost("signin")]
        public async Task<object> ApiSignIn(
                [FromBody] SignInCredentials creds) {
            SignInResult result = await SignInManager.PasswordSignInAsync(
                creds.Email, creds.Password, true, true);
            return new { success = result.Succeeded  };
        }

        [HttpPost("signout")]
        public async Task<IActionResult> ApiSignOut() {
            await SignInManager.SignOutAsync();
            return Ok();
        }
    }

    public class SignInCredentials {
        [Required]
        public string Email { get; set; }
        [Required]
        public string Password { get; set; }
    }
}
```

There are two action methods, `ApiSignIn` and `ApiSignOut`, both of which handle POST requests. The `ApiSignIn` method receives an object with `Email` and `Password` properties, which are used to sign the user into the application with the sign-in manager's `PasswordSignInAsync` method. The `ApiSignOut` method signs the user out of the application using the password manager's `SignOutAsync` method.

These are the same methods for signing in and out that I used when creating custom HTML-based workflows, and the difference is in the responses. When signing in, the API client receives an object with a `success` property, which indicates whether the sign in was successful, like this:

```
...
{ "success": true }
...
```

There is no standard way to deal with API client authentication, but the advantage of this approach is that the success or failure of a sign-in attempt is defining in an object that can contain additional data, which I'll use later in this chapter when I introduce bearer token authentication.

Enabling CORS

Cross-Origin Resource Sharing (CORS) is a security mechanism to restrict JavaScript code from making requests to a domain other than the one that served the HTML page that contains it. This wasn't an issue in earlier examples because the JavaScript client was delivered by the ASP.NET Core server, so the requests to the API controller were to the same domain. For this example, I am going to use a web server running on a different port, which is sufficiently different for CORS to block the request. Add the statements shown in Listing 12-15 to configure CORS so that the requests in this section will be allowed.

Listing 12-15. Adding a CORS Policy in the Startup.cs File in the IdentityApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using IdentityApp.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity.UI.Services;
using IdentityApp.Services;

namespace IdentityApp {

    public class Startup {

        public Startup(IConfiguration config) => Configuration = config;

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {

            // ...statements omitted for brevity...

            services.ConfigureApplicationCookie(opts => {
                opts.LoginPath = "/Identity/SignIn";
                opts.LogoutPath = "/Identity/SignOut";
                opts.AccessDeniedPath = "/Identity/Forbidden";
                opts.Events.DisableRedirectionForApiClients();
            });

            services.AddCors(opts => {
                opts.AddDefaultPolicy(builder => {
                    builder.WithOrigins("http://localhost:5100")
                        .AllowAnyHeader()
                        .AllowAnyMethod()
                        .AllowCredentials();
                });
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseRouting();
```

```
            app.UseCors();
            app.UseAuthentication();
            app.UseAuthorization();

            app.UseEndpoints(endpoints => {
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });

            app.SeedUserStoreForDashboard();
        }
    }
}
```

The AddCors method defines the CORS policy for the application. The policy is defined using the options pattern, and I use the AddDefaultPolicy method to change the default policy to allow requests that originate from JavaScript code loaded from port 5100 on the local host, with any request header, method, and credentials. The UseCors method adds middleware to the request pipeline to apply the CORS policy.

Adding Authentication to the JavaScript Client

To sign in and out of the application, the JavaScript client has to send POST requests to the action methods defined by the API controller. Add the code shown in Listing 12-16 to the network.js file to define functions that send the required requests.

Listing 12-16. Adding Functions in the network.js File in the wwwroot/js Folder

```
const baseUrl = "https://localhost:44350/api/data";
const authUrl = "https://localhost:44350/api/auth";

const baseRequestConfig = {
    credentials: "include"
}

export const signIn = async function (email, password, callback, errorHandler) {
    const response = await fetch(`${authUrl}/signin`, {
        ...baseRequestConfig,
        method: "POST",
        body: JSON.stringify({ email, password }),
        headers: { "Content-Type": "application/json" }
    });
    processResponse(response, async () =>
        callback(await response.json()), errorHandler);
}

export const signOut = async function (callback) {
    const response = await fetch(`${authUrl}/signout`, {
        ...baseRequestConfig,
        method: "POST"
    });
    processResponse(response, callback, callback);
}
```

```
export const loadData = async function (callback, errorHandler) {
    const response = await fetch(baseUrl, {
        ...baseRequestConfig,
        redirect: "manual"
    });
    processResponse(response, async () =>
        callback(await response.json()), errorHandler);
}

export const createProduct = async function (product, callback, errorHandler) {
    const response = await fetch(baseUrl, {
        ...baseRequestConfig,
        method: "POST",
        body: JSON.stringify(product),
        headers: {
            "Content-Type": "application/json"
        }
    });
    processResponse(response, callback, errorHandler);
}

export const deleteProduct = async function (id, callback, errorHandler) {
    const response = await fetch(`${baseUrl}/${id}`, {
        ...baseRequestConfig,
        method: "DELETE"
    });
    processResponse(response, callback, errorHandler);
}

function processResponse(response, callback, errorHandler) {
    if (response.ok) {
        callback();
    } else {
        errorHandler(response.status);
    }
}
```

The Fetch API that I am using to make JavaScript HTTP requests won't process cookies unless the request is configured with the `credentials` property set to `include`. This property must be set on all requests, so I have defined an object with the required setting and assigned it to a constant named `baseRequestConfig`, which I have incorporated into the other requests using the JavaScript destructuring feature. I have also defined two new functions: `signIn` is called to sign into the application, and as you might expect, the `signOut` function signs out of the application.

In Listing 12-17, I have added some new content to the HTML generated by the JavaScript client that signs a user in and out of the application. The user's credentials are hard-coded into the application, which should not be done for real projects but is sufficient for this example.

Listing 12-17. Adding Authentication Support in the client.js File in the wwwroot/js Folder

```
import * as network from "./network.js";

const columns = ["ID", "Name", "Category", "Price"];
let tableBody;
let errorElem;

// ...functions omitted for brevity...

function createContent() {
    const targetElement = createStructure();
    createAuthPrompt(targetElement);
    const table = targetElement.make("table");
    table.classList.add("table", "table-sm", "table-striped", "table-bordered");
    const headerRow = table.make("thead", "tr");
    columns.concat([""]).forEach(col => {
        const th = headerRow.make("th");
        th.innerText = col;
    });
    tableBody = table.make("tbody");
    const footerRow = table.make("tfoot", "tr");
    footerRow.make("td");
    columns.filter(col => col != "ID").forEach(col => {
        const input = footerRow.make("td", "input");
        input.name = input.id = col;
        input.placeholder = `Enter ${col.toLowerCase()}`;
    });
    const button = footerRow.make("td", "button");
    button.classList = "btn btn-sm btn-success";
    button.innerText = "Add";
    button.addEventListener("click", async () => {
        const product = {};
        columns.forEach(col => product[col] = document.getElementById(col)?.value);
        await network.createProduct(product, populateTable, showError);
    });
}

function createAuthPrompt(targetElement) {
    let signedIn = false;
    const container = targetElement.make("div");
    container.classList.add("m-2", "p-2", "text-center");
    const status = container.make("span");
    status.innerText = "Not signed in";
    const button = container.make("button");
    button.classList.add("btn", "btn-sm", "btn-secondary", "m-2");
    button.innerText = "Sign In";
    button.addEventListener("click", async () => {
        if (!signedIn) {
            await network.signIn("alice@example.com", "mysecret",
                response => {
                    if (response.success == true) {
```

```
                    signedIn = true;
                    status.innerText = "Signed in";
                    button.innerText = "Sign Out";
                    populateTable();
                }
            }, showError);
        } else {
            await network.signOut(() => {
                signedIn = false;
                status.innerText = "Signed out";
                button.innerText = "Sign In";
                createTableContents([]);
                populateTable();
            });
        }

    });
}

// ...functions omitted for brevity...

document.addEventListener("DOMContentLoaded", () => {
    createContent();
    populateTable();
})
```

The new function displays an indicator for the sign-in status and a button that calls the `signIn` and `signOut` functions when clicked. As with the existing code in this file, the emphasis is on conciseness, and the important parts of this example are the network requests made by the JavaScript client.

This example relies on delivering the JavaScript code to the browser without using a Razor Page or view. To that end, add an HTML filename `index.html` to the `wwwroot` folder with the content shown in Listing 12-18.

Listing 12-18. The Contents of the index.html File in the wwwroot Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>Identity App</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div id="target" class="m-3">Loading JavaScript client...</div>
    <script type="module" src="/js/client.js"></script>
</body>
</html>
```

Testing the Authentication API

Restart ASP.NET Core. Open a new command prompt and run the command shown in Listing 12-19 in the `wwwroot` folder to start new web server. This command relies on Node.js, which you installed in Chapter 2. If you have not installed Node.js, then you do so now.

Listing 12-19. Starting a Web Server in the wwwroot Folder

```
npx http-server -p 5100 -c-1
```

This command downloads and executes the JavaScript `http-server` package, which is a light-weight HTTP server. The arguments tell the server to listen for requests on port 5100 and to disable caching, which ensures that any changes you make to the JavaScript client will take effect the next time the browser is reloaded.

Open a new browser tab using the Guest feature so that any existing cookies won't be used. Request `http://localhost:5100`, which will send a request to the new server for the contents of the `index.html` file. The JavaScript client won't be able to populate the table with data because the ASP.NET Core API controller will return a 401 response, indicating that the request has not been authorized. Click the Sign In button, and the client will sign in and receive an ASP.NET Core cookie in the response. The request for the product data is sent again, this time with the new cookie, and the table is populated, as shown in Figure 12-4. Click the button again to sign out, and you will see another 401 error.

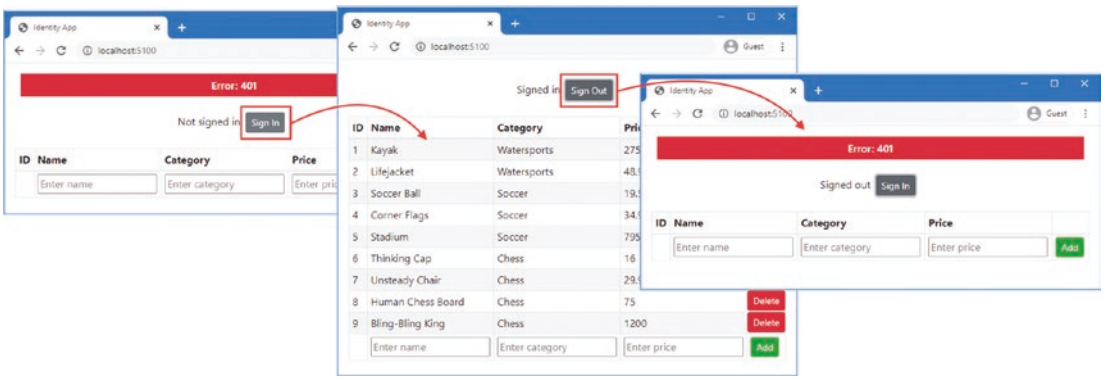

Figure 12-4. *Signing in and out with the API*

Using Bearer Tokens

You cannot rely on cookies if your API supports clients that are not browsers. Bearer tokens are strings that are provided to the client during the sign-in process and then included as a header in subsequent requests. This is essentially the same mechanism as for cookies, but the client takes responsibility for receiving the token and including it in future requests instead of relying on the browser to do the work.

When a client authenticates with the API, the response they receive will include the existing `success` property, plus a `token` that includes the value that will be included in future requests, like this:

```
...
{
  "success": true,
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9"
}
...
```

The token value is included in using the Authorization HTTP header in this format:

```
...
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
...
```

I have shortened the token to make the formatting obvious, and real tokens are much longer. Tokens are not provided when authentication fails, in which case the response will contain just the success property, like this:

```
...
{
  "success": false
}
...
```

Configuring ASP.NET Core for JWT Bearer Tokens

Microsoft provides support for dealing with a specific type of bearer token: the JSON Web Token (JWT). This type of token is specified by RFC 7519 and is used to securely describe a set of claims. The details of JWT are not important for this chapter, where the token is simply given to the client during sign-in and included in subsequent requests. Use a PowerShell command to run the command shown in Listing 12-20 in the IdentityApp folder to install the Microsoft package that supports JWT bearer tokens.

Listing 12-20. Installing a Package

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer --version 5.0.0
```

Add the configuration settings shown in Listing 12-21 to the appsettings.json file. These are the settings that will be used to generate and validate bearer tokens.

Listing 12-21. Adding Configuration Settings in the appsettings.json File

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "AppDataConnection": "Server=(localdb)\\MSSQLLocalDB;Database=IdentityAppData;Multiple
    ActiveResultSets=true",
    "IdentityConnection": "Server=(localdb)\\MSSQLLocalDB;Database=IdentityAppUserData;
    MultipleActiveResultSets=true"
  },
  "BearerTokens": {
    "ExpiryMins": "60",
```

```
    "Key": "mySuperSecretKey"
  }
}
```

The ExpiryMins property defines the period before tokens will expire. The Key property defines the key that is used to sign and validate tokens.

Listing 12-22 shows the changes to the Startup class that configure the application to support authentication with bearer tokens.

Listing 12-22. Enabling Token Authentication in the Startup.cs File in the IdentityApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using IdentityApp.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity.UI.Services;
using IdentityApp.Services;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

namespace IdentityApp {

    public class Startup {

        public Startup(IConfiguration config) => Configuration = config;

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {

            // ...statements omitted for brevity...

            services.AddAuthentication()
                .AddFacebook(opts => {
                    opts.AppId = Configuration["Facebook:AppId"];
                    opts.AppSecret = Configuration["Facebook:AppSecret"];
                })
                .AddGoogle(opts => {
                    opts.ClientId = Configuration["Google:ClientId"];
                    opts.ClientSecret = Configuration["Google:ClientSecret"];
                })
                .AddTwitter(opts => {
                    opts.ConsumerKey = Configuration["Twitter:ApiKey"];
                    opts.ConsumerSecret = Configuration["Twitter:ApiSecret"];
                    opts.RetrieveUserDetails = true;
```

```
                }).AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, opts => {
                    opts.TokenValidationParameters.ValidateAudience = false;
                    opts.TokenValidationParameters.ValidateIssuer = false;
                    opts.TokenValidationParameters.IssuerSigningKey
                        = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(
                            Configuration["BearerTokens:Key"]));
                });

            // ...statements omitted for brevity...
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            // ...statements omitted for brevity...
        }
    }
}
```

The AddJwtBearer extension method adds an authentication handler that will use JWT tokens and use them to authenticate requests. JWT tokens are intended to securely describe claims between two parties, and there are lots of features to support that goal, both in terms of the data that a token contains and how that data is validated. When using JWT tokens to authenticate clients for an ASP.NET Core API controller, the role of the token is much simpler because the JavaScript client doesn't process the contents of the token and just treats it as an opaque block of data that is included in HTTP requests to identify the client to ASP.NET Core.

This means that I need only the most basic token features. I use the options pattern to set the ValidateAudience and ValidateIssuer properties to false, which reduces the amount of data I have to put into the token later. What is important is the ability to validate the cryptographic signature that tokens include, so I read the secret key from the configuration service and apply it using the options pattern.

```
...
opts.TokenValidationParameters.IssuerSigningKey = new SymmetricSecurityKey(
    Encoding.UTF8.GetBytes(Configuration["BearerTokens:Key"]));
...
```

Updating the API Authentication Controller

The authentication handler set up by the AddJwtBearer method will validate tokens, but the application is responsible for generating them. Listing 12-23 revises the controller that signs API clients into the application so that tokens are used instead of cookies.

Listing 12-23. Using Tokens in the ApiAuthController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using SignInResult = Microsoft.AspNetCore.Identity.SignInResult;
using Microsoft.Extensions.Configuration;
using System.IdentityModel.Tokens.Jwt;
using Microsoft.IdentityModel.Tokens;
using System.Linq;
```

```
using System;
using System.Text;

namespace IdentityApp.Controllers {

    [ApiController]
    [Route("/api/auth")]
    public class ApiAuthController: ControllerBase {
        private SignInManager<IdentityUser> SignInManager;
        private UserManager<IdentityUser> UserManager;
        private IConfiguration Configuration;

        public ApiAuthController(SignInManager<IdentityUser> signMgr,
                UserManager<IdentityUser> usrMgr,
                IConfiguration config) {
            SignInManager = signMgr;
            UserManager = usrMgr;
            Configuration = config;
        }

        [HttpPost("signin")]
        public async Task<object> ApiSignIn(
                [FromBody] SignInCredentials creds) {
            IdentityUser user = await UserManager.FindByEmailAsync(creds.Email);
            SignInResult result = await SignInManager.CheckPasswordSignInAsync(user,
                creds.Password, true);
            if (result.Succeeded) {
                SecurityTokenDescriptor descriptor = new SecurityTokenDescriptor {
                    Subject = (await SignInManager.CreateUserPrincipalAsync(user))
                        .Identities.First(),
                    Expires = DateTime.Now.AddMinutes(int.Parse(
                        Configuration["BearerTokens:ExpiryMins"])),
                    SigningCredentials = new SigningCredentials(
                        new SymmetricSecurityKey(Encoding.UTF8.GetBytes(
                            Configuration["BearerTokens:Key"])),
                            SecurityAlgorithms.HmacSha256Signature)
                };
                JwtSecurityTokenHandler handler = new JwtSecurityTokenHandler();
                SecurityToken secToken = new JwtSecurityTokenHandler()
                    .CreateToken(descriptor);
                return new { success = true, token = handler.WriteToken(secToken)};
            }
            return new { success = false  };
        }

        //[HttpPost("signout")]
        //public async Task<IActionResult> ApiSignOut() {
        //    await SignInManager.SignOutAsync();
        //    return Ok();
        //}
    }
```

```
    public class SignInCredentials {
        [Required]
        public string Email { get; set; }
        [Required]
        public string Password { get; set; }
    }
}
```

The sign-in process has two key steps. The first step is to validate the password provided by the user. To do this without signing the user into the application, the sign-in manager's `CheckPasswordSignInAsync` method is used. This method operates on `IdentityUser` objects, which are obtained from the store from the user manager.

```
...
IdentityUser user = await UserManager.FindByEmailAsync(creds.Email);
SignInResult result = await SignInManager.CheckPasswordSignInAsync(user,
    creds.Password, true);
...
```

If the correct password has been provided, a token is created and sent back in the response. A `SecurityTokenDescriptor` object is created with the properties described in Table 12-5.

Table 12-5. *The SecurityTokenDescriptor Properties Used to Generate a Token*

Name	Description
`Subject`	This property is assigned the `ClaimsIdentity` object whose claims will be included in the token.
`Expires`	This property is used to specify the `DateTime` that determines when the token expires. The token won't be validated if a client presents it after this point.
`SigningCredentials`	This property is used to specify the algorithm and the secret key that will be used to sign the token.

The `SecurityTokenDescriptor` object is used to create a token using the `CreateToken` method defined by the `JwtSecurityTokenHandler` class, which is written as a string using the `WriteToken` method and sent in the response. Notice that I have commented out the `ApiSignOut` method. There is no sign-out process when using tokens, and the client simply discards the token if it is no longer required.

For quick reference, Table 12-6 describes the sign-in manager methods used to support bearer token authentication.

Table 12-6. *The SignInManager<IdentityUser> Methods to Support Bearer Tokens*

Name	Description
`CheckPasswordSignInAsync(user, password, lockout)`	This method checks a password for an `IdentityUser` object without signing the user into the application. The `lockout` argument specifies whether an incorrect password will count toward a lockout. Lockouts are described in Chapter 9.
`CreateUserPrincipalAsync(user)`	This method creates a `ClaimsPrincipal` object from an `IdentityUser` object, which is used as the data for the bearer token.

Specifying Token Authentication in the API Controller

The final server-side change is to specify that requests for the API controller should be authenticated using the token handler, as shown in Listing 12-24.

Listing 12-24. Specifying Token Authentication in the ValuesController.cs File in the Controllers Folder

```
using IdentityApp.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authentication.JwtBearer;

namespace IdentityApp.Controllers {

    [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
    [ApiController]
    [Route("/api/data")]
    public class ValuesController : ControllerBase {

        // ...statements omitted for brevity...
    }

    public class ProductBindingTarget {
        [Required]
        public string Name { get; set; }

        [Required]
        public decimal Price { get; set; }

        [Required]
        public string Category { get; set; }
    }

}
```

The `AuthenticationSchemes` argument to the `Authorize` attribute is used to specify the same name that was used to set up the handler in the `Startup` class.

Updating the JavaScript Client

Now that the server-side pieces are in place, I can update the JavaScript client to receive and use bearer tokens, as shown in Listing 12-25.

Listing 12-25. Using Tokens in the network.js File in the wwwroot/js Folder

```
const baseUrl = "https://localhost:44350/api/data";
const authUrl = "https://localhost:44350/api/auth";

const baseRequestConfig = {
```

```
    credentials: "include"
}

export const signIn = async function (email, password, callback, errorHandler) {
    const response = await fetch(`${authUrl}/signin`, {
        ...baseRequestConfig,
        method: "POST",
        body: JSON.stringify({ email, password }),
        headers: {
            "Content-Type": "application/json"
        }
    });

    if (response.ok) {
        let responseData = await response.json();
        if (responseData.success) {
            baseRequestConfig.headers = {
                "Authorization": `Bearer ${responseData.token}`
            }
        }
        processResponse(response, async () =>
            callback(responseData, errorHandler));
        return;
    }
    processResponse({ ok: false, status: "Auth Failed" }, async () =>
        callback(responseData), errorHandler);
}

export const signOut = async function (callback) {
    //const response = await fetch(`${authUrl}/signout`, {
    //    ...baseRequestConfig,
    //    method: "POST"
    //});
    baseRequestConfig.headers = {};
    processResponse({ ok: true }, callback, callback);
}

export const loadData = async function (callback, errorHandler) {
    const response = await fetch(baseUrl, {
        ...baseRequestConfig,
        redirect: "manual"
    });
    processResponse(response, async () =>
        callback(await response.json()), errorHandler);
}

export const createProduct = async function (product, callback, errorHandler) {
    const response = await fetch(baseUrl, {
        ...baseRequestConfig,
        method: "POST",
        body: JSON.stringify(product),
```

```
        headers: {
            ...baseRequestConfig.headers,
            "Content-Type": "application/json"
        }
    });
    processResponse(response, callback, errorHandler);
}

export const deleteProduct = async function (id, callback, errorHandler) {
    const response = await fetch(`${baseUrl}/${id}`, {
        ...baseRequestConfig,
        method: "DELETE"
    });
    processResponse(response, callback, errorHandler);
}

function processResponse(response, callback, errorHandler) {
    if (response.ok) {
        callback();
    } else {
        errorHandler(response.status);
    }
}
```

The `signIn` function processes successful responses to get the token, which is added as a header to the object used to configure the requests. The `signOut` function has been updated to remove the header from the configuration object, which discards the token when the user wants to sign out. An adjustment is required to the `createProduct` function so that the header required to set the content type of the request is added to the headers defined by the base configuration object.

Testing Token Authentication

Restart ASP.NET Core and restart the HTTP server using the command shown in Listing 12-19. Once both servers have started, use your browser's guest mode to request `http://localhost:5100`, which will ensure that ASP.NET Core authentication cookies from earlier examples are not used in the authentication process. Even though the mechanism used to authenticate the JavaScript client has changed, the user experience remains the same. The initial request fails because it doesn't include a token. Clicking the Sign In button sends a request to obtain a token, which is used to make another request for the data. Clicking the Sign Out button discards the token, causing the data request to fail, as shown in Figure 12-5.

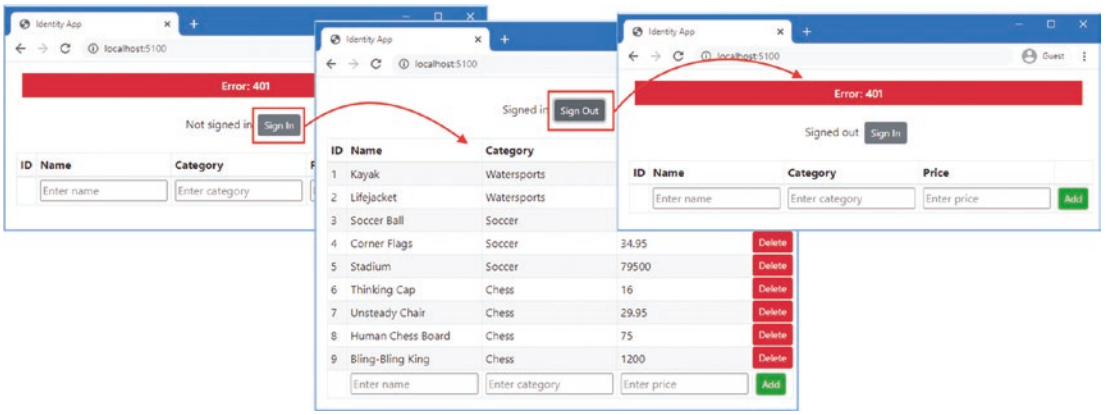

Figure 12-5. *Authenticating using a bearer token*

Summary

In this chapter, I explained how to authenticate API clients. I explained how to take advantage of the standard ASP.NET Core authentication cookie for quick and easy API authentication. Not all projects can use this approach, so I also explained how to obtain a cookie directly and how to use bearer tokens, which are useful for clients that don't support cookies. In the next part of this book, I revisit the features provided by the Identity API and explain how they work in detail.