**Chapter 3. Implement Azure security**

Regardless of the application, most of them have a standard requirement—protect the information that it manages. With regard to security, you need to think about the five dimensions of information security awareness: integrity, availability, confidentiality, authorization, and no-repudiation. Each of these dimensions is useful for evaluating the different risks and the countermeasures that you need to implement for mitigating the associated risks.

Implementing the appropriate security mechanism on your application can be tedious and potentially error-prone. Azure offers several mechanisms for adding security measures to your applications, controlling the different security aspects for accessing your data, and controlling the services that depend on your applications.

**Skills covered in this chapter:**

* [Skill 3.1: Implement user authentication and authorization](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev1sec1)
* [Skill 3.2: Implement secure cloud solutions](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev1sec2)

**Skill 3.1: Implement user authentication and authorization**

When a user wants to access your application, the user needs to prove that he or is the person he or she claims to be. Authentication is the action that the user performs to prove his or her identity. The user proves his or her identity using information known only to the user. An authentication system needs to address how to protect that information so only the appropriate user can access it while nobody else—not even the authorization system—can access it. A solution for this problem is to allow the user to access his or her data by using two different mechanisms for proving his or her identity—information that only the user knows and showing something, a token, only the user has. This approach is known as *multifactor authentication*.

Azure provides a secure mechanism for integrating authentication into your applications. You can use single-factor or multifactor authentication systems without worrying about the intricate details of implementing this kind of system.

Authenticating users before they can access your application is only part of the equation. Once your users have been authenticated, you need to decide if any user can access any part of your application, or if some parts of your application are restricted. The authorization controls which actions or sections the user can perform once he or she has been authorized.

**This skill covers how to**

* [Implement OAuth2 authentication](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev2sec1)
* [Create and implement shared access signatures](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev2sec2)
* [Register apps and use Azure Active Directory to authenticate users](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev2sec3)
* [Control access to resources by using role-based access control (RBAC)](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev2sec4)

**Implement OAuth2 authentication**

The authentication process requires the user to provide evidence that the user is the person he or she claims to be. In the real world, you can find multiple examples of authentication; for example, every time that you show your driver’s license to a police officer, you are actually authenticating against the police officer. In the digital world, this authentication happens by providing some information that only you know, such as a secret word (a password), a digital certificate, or any kind of token that only you possess.

You have a range of options for implementing such an authentication mechanism in your application. Each implementation has its pros and cons, and the appropriate authentication mechanism depends on the level of security that you require for your application.

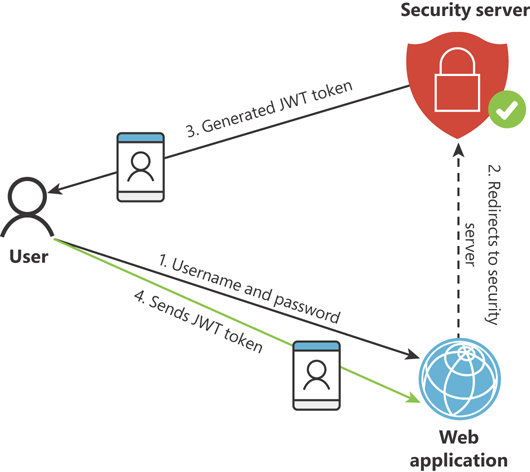
The most basic way of authenticating a user is form-based authentication. When you use this mechanism, you need to program a web form that asks the user for a username and a password. Once the user submits the form, the information in the form is compared to the values stored in your storage system. This storage system can be a relational database, a NoSQL database, or even a simple file with different formats stored on a server. If the information provided by the user matches the information stored in your system, the application sends a cookie to the user’s browser. This cookie stores a key or some type of ID for authenticating subsequent requests to access your application without repeatedly asking the user for his or her username and password.

One of the most significant drawbacks of using form-based authentication is the authentication mechanism’s dependency on cookies. Another inconvenience is that this is stateful, which requires that your server keeps an authentication session for tracking the activity between the server and the client. This dependency on cookies and authentication session management makes it more difficult to scale solutions using form-based authentication. One additional point to consider is that cookies don’t work well (or it’s challenging to work with them) on mobile apps. Fortunately, there are alternatives to form-based authentication that are more suitable for the requirements that have mobile or IoT scenarios; also, there are alternatives that can improve the scalability of your web application.

Token-based authentication is the most extended authentication mechanism for environments and scenarios that require high scalability or do not support the usage of cookies. Token-based authentication consists of a signed token that your application uses for authenticating requests and granting access to the resources in your application. The token does not contain the username and password of your user. Instead, the token stores some information about the authenticated user that your server can use for granting access to your application’s resources.

When you use token-based authentication, you follow a workflow similar to the one shown in [Figure 3-1](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig01):

1. An unauthenticated user connects to your web application.
2. Your web application redirects the user to the login page. This login page can be provided by your web application acting as a security server or by an external security server.
3. The security server validates the information provided by the user—typically, the username and password—and generates a JWT token.
4. The security server sends the JWT token to the user. The browser or mobile app that the user used to connect to your application is responsible for storing this JWT token for reusing it in the following requests.
5. The browsers or mobile app provides the JWT token to your web application on each following request.



**Figure 3-1** Basic workflow of token-based authentication

There are several token implementations, but the most extended one is JSON Web Token or JWT. A JWT token consists of

* **Header** The header contains the name of the algorithm used for signing the token.
* **Body or Payload** The body or payload contains different information about the token and the purpose of this token. The body contains several standard fields or claims that are defined in the RFC 7519 and any other custom field that you may need for your application.
* **Cryptographic signature** This is a string that validates your token and ensures that the token has not been corrupted or incorrectly manipulated.

One of the main advantages of using token-based authentication is that you can delegate the process of managing the identity of the users to external security servers. Thanks to this delegation, you can abstract from the implementation of managing and storing JWT tokens and usernames and passwords. That is what you do when you want to allow your users to access your application by using their Facebook, Google, or Twitter accounts. Your application trusts the identification and authentication processes made by these external security servers or identity managers, and you grant access to your application based on the information stored in the JWT token provided by the security server. You still need to store some information about the user. Still, there is no need to know anything about the password or any other information that the user needed to provide to the security server for authentication.

Microsoft provides the Identity Framework for working with authentication. You can use the Identity Framework for adding token-based authentication to your application. As previously mentioned, you can implement your own token-based authentication or use an external authentication provider that performs the verification of the user’s login and password. The following example shows how to create a simple web application with Google authentication enabled. You can use a similar procedure for enabling another social login to your application, such as Facebook, Twitter, or Microsoft accounts.

1. Open Visual Studio 2019 on your computer.
2. In the welcome window of Visual Studio 2019, on the Get Started column, click Create A New Project.
3. On the Create A New Project window, on the All Languages drop-down menu, select C#.
4. In the Search For Templates text box, type **asp.net**.
5. On the result list, click ASP.NET Web Application (.NET Framework).
6. Click the Next button at the bottom-right corner of the window.
7. On the Configure Your New Project, type a Project Name, a Location, and a Solution Name for your project.
8. Click the Create button at the bottom-right corner of the window.
9. On the Create A New ASP.NET Web Application window, select MVC template from the template list in the middle of the left side of the window. MVC is for Model-View-Controller.
10. On the right side of the Create A New ASP.NET Web Application window, in the Authentication section, click the Change link.
11. On the Change Authentication window, select Individual User Accounts from the available options in the left column.
12. Click the OK button for closing the Change Authentication window.
13. Click the Create button at the bottom-right corner of the window.

At this point, you have created a basic ASP.NET web application configured for using form-based authentication. Now, you can check that the basic authentication in this application works. At this point, you are not using OAuth2 authentication:

1. Press F5 to run the project.
2. In the top-right corner of your application’s web browser, click Register.
3. On the Register form, enter an email address and password.
4. Click the Register button at the bottom of the form.

Once you have registered, you are automatically logged on, and you can log off and log in again to ensure everything works properly.

Now, use the following steps for modifying the newly created web application for adding OAuth2 authentication:

1. In the Solution Explorer, click the name of your project and press F4.
2. In the Development Server section, ensure the value of the SSL Enabled setting is set to True.
3. Copy the SSL URL below the SSL Enabled setting, and close the Properties window.
4. In the Solution Explorer, right-click the project’s name and click Properties at the bottom of the contextual menu. This opens your project’s csproj file in a new tab.
5. On your project’s csproj file tab, select the Web tab and paste the SSL URL in the Project Url text box in the Servers section.
6. Open the HomeController.cs file and add the RequireHttps attribute to the HomeController class:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg131a)

[RequireHttps]

public class HomeController : Controller

{

public ActionResult Index()

1. Create a Google Project for integrating your web application with Google’s authentication platform. You need a Google account for these steps:
   1. Log in to your Google account.
   2. Navigate to [*https://developers.google.com/identity/sign-in/web/devconsole-project*](https://developers.google.com/identity/sign-in/web/devconsole-project).
   3. Click Configure A Project.
   4. On the Configure A Project For Google Sign-In dialog box, select Create A New Project from the drop-down menu.
   5. Enter a name for your project and click Next at the bottom-left corner of the dialog box.
   6. On the Configure Your OAuth Client dialog box, type the name of your web application. This name is shown in the consent window that appears to the user during login.
   7. Click Next at the bottom-left corner of the dialog box.
   8. On the Configure Your OAuth Client dialog box, select Web Server in the Where Are You Calling From? drop-down menu.
   9. In the Authorized Redirect URIs text box, use the URL SSL that you copied in step 4 and create a redirect URI with this structure:

<YOUR\_URL\_SSL>/signin-google

Use the following example for your reference:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg132-1a)

*https://localhost:44395/signin-google*

* 1. Click Create in the bottom-left corner of the dialog box.
  2. On the You’re All Set! dialog box, click the Download Client Configuration button. Alternatively, you can copy the Client ID and Client Secret fields. You need these values in a later step.
  3. Click the API Console link at the bottom of the dialog box.
  4. On the left side of the Google Console, click Library.
  5. In the Search For APIs & Services text box, type **Google+**.
  6. In the result list, click Google+ API.
  7. In the Google+ API window, click the Enable button.

1. In the App\_Start/Startup.Auth.cs file, in the ConfigureAuth method, uncomment the following lines:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg132-2a)

app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()

{

ClientId = "",

ClientSecret = ""

});

1. Use the Client ID and Client Secret values that you copied in step 7k and assign the value to the corresponding variable in the preceding code. (Note that these items need to be placed inside quotation marks.)
2. Press F5 for running the application.
3. Click Log In in the top-left corner of the web application.
4. Under the Use Another Service To Log In option on the left side of the page, click the Google button.
5. Log in using your Google account.
6. Click the Register button. Once you are logged in with Google, you are redirected to the web application. Because your Google account does not exist on your application’s database, you get a registration form.
7. You are registered and logged in to your application using a Google account.

Once you have logged in to the application using your Google account, you can review the database and look for the new login information. You can view the database connections in the SQL Server Object Explorer. You can open the SQL Server Object Explorer from the View menu. You can see in the AspNetUsers table that there is a new entry for your new Google account login, but the PasswordHash field is empty. You can also review the AspNetUserLogins table. This table contains all the logins from external authorization providers, such as Google, that have been registered in your application.

The previous example reviewed how you can use third-party identity servers, like Google, for authenticating users in your application using the OAuth2 protocol. Now, we are going to review how to create your OAuth2 server. You usually add this type of authentication when you need these third-party applications to access some services or resources of your code. You can grant access to these applications by creating a token that authenticates the third-party application and grants access to specific parts of your HTTP service.

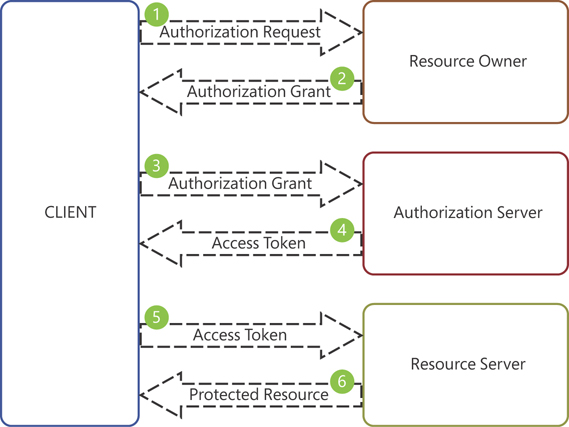
The OAuth protocol addresses the need to secure access to resources and information in your application by the third party’s process. Without OAuth, if you want to grant access to an external application to the resources of your application, you need to use a username and password. If the third-party application is compromised, then the username and password are also compromised, and your resources are exposed. The OAuth protocol defines four different roles:

* **Resource owner** This is the person or entity that can grant access to the resources. If the resource owner is a person, it can also be referred to as the user.
* **Resource server** This is the server that hosts the resources that you want to share. This server needs to be able to accept and respond to the access codes used for accessing the resource.
* **Client** This is the third-party application that needs to access the resource. The client makes the needed requests to the resource server on behalf of the resource owner. The term “client” does not necessarily imply any specific implementation, like a server, a desktop, or any other kind of device.
* **Authorization server** This is the server that issues the access token to the client for accessing the resources. The client needs to be authenticated before it can get the correct token.

[Figure 3-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig02) shows the basic authentication flow for OAuth.

As you can see in [Figure 3-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig02), the process of acquiring a token for accessing a protected resource consists of the following steps:

* **Authentication request** The client requests access to the protected resource. The resource owner, based on the privileges of the client, grants access to the client for accessing the resource. The authentication of the client can be directly done by the resource owner or preferably by the authentication server.



**Figure 3-2** OAuth basic authentication flow

* **Authentication grant** When the resource owner grants the client access to the resource, the client sends an authentication grant, which is a code or credential that represents the permission to access the resource, which has been granted by the resource owner. The client uses this authentication grant credential to request an access token to the authorization server. There are four different mechanisms for handling this authentication:
  + **Authorization code** The client instructs the resource owner to request authentication to the authentication server. Once the resource owner is authenticated, the authentication server creates an authorization code that the resource owner sends back to the client. The client uses this authorization code as the grant for requesting the access token.
  + **Implicit** Using this authentication grant flow, the authentication server does not authenticate the client. Instead, the client gets the access token without needing to authenticate to the resource server using an authentication grant. This implicit flow is a simplified authorization code flow. To improve security in this flow, the resource server uses the redirect URI provided by the client.
  + **Resource owner credentials** Instead of using an authorization code or implicit authentication, the client uses the credentials of the resource owner for authenticating against the resource server. This type of authentication grant should be used only when there is a high level of trust between the client and the resource owner.
  + **Client credentials** The client provides his or her credentials for accessing the resource. This authentication grant is useful for scenarios in which the client needs access to resources that are protected by the same authorization server as the client and are under 135the control of the client. This type of authentication grant is also useful if the resource server and the client arranged the same authorization for the resources and the client.
* **Access token** The client requests an access token from the authorization server that allows the client to access the resource on the resource server. The client sends this access token to the resource server with each request to access the resource. This access token has an expiration date. Once the access token is expired, the token is invalid, and the client needs to request another access token. To ease the process of renewing the access token, the authentication server provides two different tokens—the actual access token and a refresh token. The client uses the refresh token when it needs to renew an expired access token.
* **Protected resource** This is the resource that the client wants to access. The resource server protects the resource. The client needs to send the access token to the resource server every time it needs to access the resource.

***NEED MORE REVIEW?* THE OAUTH 2.0 AUTHORIZATION FRAMEWORK**

You can get more information about the details of how the OAuth 2.0 Authorization Framework works by reviewing the official RFC 6749 at [*https://tools.ietf.org/html/rfc6749*](https://tools.ietf.org/html/rfc6749).

The following example shows how to implement OAuth 2.0 authentication in your Web API application. In this example, you are going to create an authorization server, a resource server, and a client that can request an access token before accessing the resource. For the sake of readability, we have split the steps for implementing this example into different parts. The following steps show how to create the authorization server:

1. Open Visual Studio 2019.
2. Click File > New > Project.
3. On the Create A New Project window, on the All Languages drop-down menu, select C#.
4. On the Search For Templates text box type **asp.net**.
5. On the result list, click ASP.NET Web Application (.NET Framework).
6. Click the Next button at the bottom-right corner of the window.
7. In the Configure Your New Project window, type a Project Name, a Location, and a Solution Name for your project.
8. Click the Create button at the bottom-right corner of the window.
9. In the Create A New ASP.NET Web Application window, click the MVC template.
10. Click the Change link in the Authentication section on the right side of the window.
11. On the Change Authentication window, click the Individual User Accounts option.
12. Click the OK button on the Change Authentication window.
13. Click the Create button on the Create A New ASP.NET Web Application window.
14. In Visual Studio, open the file at App\_Start > Startup.Auth.cs, and add the following line to the beginning of the file:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg136-1a)

using Microsoft.Owin.Security.OAuth;

1. Add the code shown in [Listing 3-1](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex01) to the Startup.Auth.cs file. You need to add this code to the ConfigureAuth() method, after the line

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg136-2a)

app.UseTwoFactorRememberBrowserCookie(DefaultAuthenticationTypes.

TwoFactorRememberBrowserCookie);

1. Ensure the following using statements exist in the Startup.Auth.cs file for avoiding compilation errors:
   * using System;
   * using Microsoft.AspNet.Identity;
   * using Microsoft.AspNet.Identity.Owin;
   * using Owin;
   * using Microsoft.Owin;
   * using Microsoft.Owin.Security.Cookies;
   * using Microsoft.Owin.Security.OAuth;
   * using Microsoft.Owin.Security.Infrastructure;
   * using AuthorizationServer.Constants;
   * using System.Threading.Tasks;
   * using System.Collections.Concurrent;
   * using System.Security.Claims;
   * using System.Security.Principal;
   * using System.Linq;
   * using *<your\_project's\_name>*.Models;

**Listing 3-1** Adding OAuth Authorization Server

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-1a)

// C#. ASP.NET.

//Setup the Authorization Server

app.UseOAuthAuthorizationServer(new OAuthAuthorizationServerOptions

{

AuthorizeEndpointPath = new PathString(Paths.AuthorizePath),

TokenEndpointPath = new PathString (Paths.TokenPath),

ApplicationCanDisplayErrors = true,

#if DEBUG

AllowInsecureHttp = true,

#endif

Provider = new OAuthAuthorizationServerProvider

{

OnValidateClientRedirectUri = ValidateClientRedirectUri,

OnValidateClientAuthentication = ValidateClientAuthentication,

OnGrantResourceOwnerCredentials = GrantResourceOwnerCredentials,

OnGrantClientCredentials = GrantClientCredentials

},

// The authorization code provider is the object in charge of creating and receiving

// the authorization code.

AuthorizationCodeProvider = new AuthenticationTokenProvider

{

OnCreate = CreateAuthenticationCode,

OnReceive = ReceiveAuthenticationCode,

},

// The refresh token provider is in charge in creating and receiving refresh

// token.

RefreshTokenProvider = new AuthenticationTokenProvider

{

OnCreate = CreateRefreshToken,

OnReceive = ReceiveRefreshToken,

}

});

//Protect the resources on this server.

app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions

{

});

This code configures the OAuth Authentication Server by using the UseOAuthAuthorizationServer() method. This method accepts an OAuthAuthorizationServerOptions object for configuring several useful endpoints:

* + **AuthorizeEndpointPath** The authorize endpoint is the path in the authorization server to which the client application redirects the user-agent to obtain the user or resource owner’s consent to access the resource. With this consent, the client application can request an access token.
  + **TokenEndpointPath** This is the path in the authorization server that the client uses to obtain an access token. If the client is configured with a client secret, the client needs to provide this client secret on the request for obtaining a new token.
  + **AllowInsecureHttp** This setting allows the client to make requests to the authorize and token endpoints by using HTTP URIs instead of HTTPS URIs.
  + **Provider** Your authorization server application needs to provide the needed delegated methods for processing the different events that arise during the OAuth authorization flow. You can do this by implementing the OAuthAuthorizationServerProvider interface or using the default implementation provided by the OAuthAuthorizationServerProvider object. In this example, you use the OAuthAuthorizationServerProvider object and provide four delegate functions for the different events. [Listings 3-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex02) to [3-5](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex05) show the different delegate methods that you use for the events managed by this provider.
  + **AuthorizationCodeProvider** When the authorization server authenticates the client, the server needs to send an authorization code to the server. This provider manages the events that arise during the management of the authentication code. [Listings 3-6](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex06) and [3-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex07) show the delegate methods that manage the events of creating or receiving a code.
  + **RefreshTokenProvider** This object controls the events that happen when the client requests a refresh of an access token. [Listings 3-8](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex08) and [3-9](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex09) show the delegate methods that control the events of creating and receiving a request of refreshing an access token.

1. Add the content from [Listings 3-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex02) to [3-9](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex09) to the Startup.Auth.cs file. Add these methods to the Startup class. The implementation of these delegates is not suitable for production environments. For example, the validation of the client redirect URI, and the authentication of the clients are based on a hard-coded value stored in the Client class. In a real-world scenario, you should have these entities stored in a database. In this example, the creation of the access token, shown in [Listing 3-4](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex04), is stored in an in-memory dictionary. In a real-world scenario, you should save in a database the access tokens that you grant to the clients.

**Listing 3-2** OnValidateClientRedirectUri delegate

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-2a)

**// C#. ASP.NET.**

private Task ValidateClientRedirectUri(OAuthValidateClientRedirectUriContext context)

{

if (context.ClientId == Clients.Client1.Id)

{

context.Validated(Clients.Client1.RedirectUrl);

}

else if (context.ClientId == Clients.Client2.Id)

{

context.Validated(Clients.Client2.RedirectUrl);

}

return Task.FromResult(0);

}

**Listing 3-3** OnValidateClientAuthentication delegate

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-3a)

**// C#. ASP.NET.**

private Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext

context)

{

string clientId;

string clientSecret;

if (context.TryGetBasicCredentials(out clientId, out clientSecret) ||

context.TryGetFormCredentials(out clientId, out clientSecret))

{

if (clientId == Clients.Client1.Id && clientSecret == Clients.Client1.

Secret)

{

context.Validated();

}

else if (clientId == Clients.Client2.Id && clientSecret == Clients.

Client2.Secret)

{

context.Validated();

}

}

return Task.FromResult(0);

}

**Listing 3-4** OnGrantResourceOwnerCredentials delegate

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-4a)

**// C#. ASP.NET.**

private Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext

context)

{

ClaimsIdentity identity = new ClaimsIdentity(new GenericIdentity(context.

UserName, OAuthDefaults.AuthenticationType), context.Scope.Select(x =>

new Claim("urn:oauth:scope", x)));

context.Validated(identity);

return Task.FromResult(0);

}

**Listing 3-5** OnGrantClientCredentials delegate

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-5a)

**// C#. ASP.NET.**

private Task GrantClientCredentials(OAuthGrantClientCredentialsContext context)

{

var identity = new ClaimsIdentity(new GenericIdentity(context.ClientId,

OAuthDefaults.AuthenticationType), context.Scope.Select(x =>

new Claim("urn:oauth:scope", x)));

context.Validated(identity);

return Task.FromResult(0);

}

**Listing 3-6** Authorization code for OnCreate delegate

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-6a)

**// C#. ASP.NET.**

private void CreateAuthenticationCode(AuthenticationTokenCreateContext context)

{

context.SetToken(Guid.NewGuid().ToString("n") + Guid.NewGuid().

ToString("n"));

authenticationCodes[context.Token] = context.SerializeTicket();

}

**Listing 3-7** Authorization code for OnReceive delegate

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-7a)

**// C#. ASP.NET.**

private void ReceiveAuthenticationCode(AuthenticationTokenReceiveContext context)

{

string value;

if (\_authenticationCodes.TryRemove(context.Token, out value))

{

context.DeserializeTicket(value);

}

}

**Listing 3-8** Refresh token for OnCreate delegate

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-8a)

**// C#. ASP.NET.**

private void CreateRefreshToken(AuthenticationTokenCreateContext context)

{

context.SetToken(context.SerializeTicket());

}

**Listing 3-9** Refresh token for OnReceive delegate

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-9a)

**// C#. ASP.NET.**

private void ReceiveRefreshToken(AuthenticationTokenReceiveContext context)

{

context.DeserializeTicket(context.Token);

}

1. Add the following private property to the Startup class in the Startup.Auth.cs file:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg141a)

private readonly ConcurrentDictionary<string, string> \_authenticationCodes =

new ConcurrentDictionary<string, string>(StringComparer.Ordinal);

1. On the Solution Explorer window, add a new folder to your project called Constants.
2. On the Solution Explorer window, right-click the Constants folder and click Add > New Item.
3. On the New Item window, on the tree control on the left side of the window, click Installed > Visual C# > Code.
4. Click the template named Class.
5. At the bottom of the Add New Item window, type **Clients.cs** in the Name text box.
6. Click the Add button in the bottom-right corner of the window.
7. Replace the content of the Clients.cs file with the content in [Listing 3-10](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex10). Change the namespace to match your project’s name.

**Listing 3-10** Clients.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-10a)

// C#. ASP.NET.

namespace <YOUR\_PROJECT'S\_NAME>.Constants

{

public class Clients

{

public readonly static Client Client1 = new Client

{

Id = "123456",

Secret = "abcdef",

RedirectUrl = Paths.AuthorizeCodeCallBackPath

};

public readonly static Client Client2 = new Client

{

Id = "78901",

Secret = "aasdasdef",

RedirectUrl = Paths.ImplicitGrantCallBackPath

};

}

public class Client

{

public string Id { get; set; }

public string Secret { get; set; }

public string RedirectUrl { get; set; }

}

}

1. On the Solution Explorer window, click your project’s name and press F4.
2. On your project’s properties window, ensure the value of SSL Enabled is set to True.
3. Copy the value of the SSL URL setting.
4. Right-click the project’s name and click the Properties menu item at the bottom of the contextual menu.
5. On the project’s properties tab in Visual Studio, click the Web element on the left side of the window.
6. In the Servers section, paste the SSL URL value that you copied in step 28 in the Project URL text box.
7. Add a new empty C# class to the Constants folder and name it **Paths.cs**. You can repeat the steps 20–24 to create a new C# class.
8. Replace the content of the file Paths.cs with the code shown in [Listing 3-11](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex11).
9. Paste the value of the SSL URL that you copied on step 28 on the following constants:
   * **AuthorizationServerBaseAddress**
   * **ResourceServerBaseAddress**
   * **ImplicitGrantCallBackPath** Ensure that you don’t delete the URI part. This constant should look like <SSL URL>/Home/SignIn.
   * **AuthorizeCodeCallBackPath** Ensure that you don’t delete the URI part. This constant should look like <SSL URL>/Manage.

**Listing 3-11** Paths.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-11a)

// C#. ASP.NET.

namespace <YOUR\_PROJECT'S\_NAME>.Constants

{

public class Paths

{

public const string AuthorizationServerBaseAddress = "https://localhost:44317";

public const string ResourceServerBaseAddress = "https://localhost:44317";

public const string ImplicitGrantCallBackPath =

"https://localhost:44317/Home/SignIn";

public const string AuthorizeCodeCallBackPath = "https://localhost:44317/Manage";

public const string AuthorizePath = "/OAuth/Authorize";

public const string TokenPath = "/OAuth/Token";

public const string LoginPath = "/Account/Login";

public const string LogoutPath = "/Account/Logout";

public const string MePath = "/api/Me";

}

}

At this point, you need to create the API Controller that manages the requests to the Authorize and Token endpoint. When you configured the Authentication Server, you used the following code snippet for setting the endpoints that the server uses for attending OAuth requests:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg143a)

app.UseOAuthAuthorizationServer(new OAuthAuthorizationServerOptions

{

AuthorizeEndpointPath = new PathString(Paths.AuthorizePath),

TokenEndpointPath = new PathString(Paths.TokenPath),

If you review the value of the parameters AuthorizePath and TokenPath in your Paths class, you can see that their values are /OAuth/Authorize and /OAuth/Token, respectively. Now, you need to create the controller that manages the requests to these endpoints.

1. In the Solution Explorer window, right-click the Controllers folders in your project, and then choose Add > Controller.
2. On the Add Scaffold window, choose MVC 5 Controller – Empty.
3. Click the Add button.
4. On the Add Controller window, type **OAuthController**.
5. Open the OAuthController.cs file and replace the content of the file with the code shown in [Listing 3-12](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex12).

**Listing 3-12** OAuthController.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-12a)

// C#. ASP.NET.

using System.Security.Claims;

using System.Web;

using System.Web.Mvc;

namespace <your\_project's\_name>.Controllers

{

public class OAuthController : Controller

{

// GET: OAuth/Authorize

public ActionResult Authorize()

{

if (Response.StatusCode != 200)

{

return View("AuthorizeError");

}

var authentication = HttpContext.GetOwinContext().Authentication;

var ticket = authentication.AuthenticateAsync("ApplicationCookie").Result;

var identity = ticket != null ? ticket.Identity : null;

if (identity == null)

{

authentication.Challenge("ApplicationCookie");

return new HttpUnauthorizedResult();

}

var scopes = (Request.QueryString.Get("scope") ?? "").Split(' ');

if (Request.HttpMethod == "POST")

{

if (!string.IsNullOrEmpty(Request.Form.Get("submit.Grant")))

{

identity = new ClaimsIdentity(identity.Claims, "Bearer", identity.

NameClaimType, identity.RoleClaimType);

foreach (var scope in scopes)

{

identity.AddClaim(new Claim("urn:oauth:scope", scope));

}

authentication.SignIn(identity);

}

if (!string.IsNullOrEmpty(Request.Form.Get("submit.Login")))

{

authentication.SignOut("ApplicationCookie");

authentication.Challenge("ApplicationCookie");

return new HttpUnauthorizedResult();

}

}

return View();

}

}

}

1. On the Solution Explorer, right-click Views > OAuth, and then select Add > View.
2. On the Add View window, on the View Name field, type **Authorize**.
3. Click Add.
4. Replace the content of the file Authorize.cshtml with the code shown in [Listing 3-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex13):

**Listing 3-13** Authorize.cshtml

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-13a)

// C#. ASP.NET.

@{

ViewBag.Title = "Authorize";

}

@using System.Security.Claims

@using System.Web

@{

var authentication = Context.GetOwinContext().Authentication;

var ticket = authentication.AuthenticateAsync("ApplicationCookie").Result;

var identity = ticket != null ? ticket.Identity : null;

var scopes = (Request.QueryString.Get("scope") ?? "").Split(' ');

}

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>@ViewBag.Title</title>

</head>

<body>

<h1>Authorization Server</h1>

<h2>OAuth2 Authorize</h2>

<form method="POST">

<p>Hello, @identity.Name</p>

<p>A third party application wants to do the following on your behalf:</p>

<ul>

@foreach (var scope in scopes)

{

<li>@scope</li>

}

</ul>

<p>

<input type="submit" name="submit.Grant" value="Grant" />

<input type="submit" name="submit.Login" value="Sign in as different user" />

</p>

</form>

</body>

</html>

1. Add another empty view named **AuthorizeError**.
2. Replace the content of the file AuthorizeError.cshtml with the code shown in [Listing 3-14](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex14):

**Listing 3-14** AuthorizeError.cshtml

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-14a)

// C#. ASP.NET.

@{

ViewBag.Title = "AuthorizeError";

}

@using System

@using System.Security.Claims

@using System.Web

@using Microsoft.Owin

@{

IOwinContext owinContext = Context.GetOwinContext();

var error = owinContext.Get<string>("oauth.Error");

var errorDescription = owinContext.Get<string>("oauth.ErrorDescription");

var errorUri = owinContext.Get<string>("oauth.ErrorUri");

}

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>@ViewBag.Title</title>

</head>

<body>

<h1>Katana.Sandbox.WebServer</h1>

<h2>OAuth2 Authorize Error</h2>

<p>Error: @error</p>

<p>@errorDescription</p>

</body>

</html>

This example only provides an implementation for the Authorize endpoint for the sake of simplicity. An authorized user in your application needs to grant access to the resources in your application explicitly. When the user grants those privileges, the application automatically creates an in-memory OAuth token that you can use to make a request to the protected resources. In a real-world scenario, this process should be separated in the two different endpoints—Authorize and Token. You should use the Token endpoint for creating or refreshing the access token issued by the authorization server.

Now that you have created and configured your authorization server, you can create the resource server. In this example, you are going to create the resource server on the same application where you implemented the authorization server. In a real-world scenario, you can use the same application, or you can use a different application deployed by a different server or Azure App Service.

1. On the Solution Explorer window, right-click the Controllers folder in your project and click Add > Controller.
2. On the Add New Scaffolded Item window, select the Web API 2 Controller — Empty template.
3. Click the Add button.
4. In the Add Controller window, type **MeController** and click the Add button.
5. Replace the content of the MeController.cs file with the code shown in [Listing 3-15](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex15). This controller is quite simple and only returns the information stored in the token that you provide to the resource server when you try to access the resource.

**Listing 3-15** MeController.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-15a)

// C#. ASP.NET.

using System.Collections.Generic;

using System.Linq;

using System.Security.Claims;

using System.Web.Http;

namespace <your\_project's\_name>.Controllers

{

[Authorize]

public class MeController : ApiController

{

// GET api/<controller>

public IEnumerable<object> Get()

{

var identity = User.Identity as ClaimsIdentity;

return identity.Claims.Select(c => new

{

Type = c.Type,

Value = c.Value

});

}

}

}

1. On the Solution Explorer window, in the App\_Start folder, rename the file WebApiConfig.cs to Startup.WebApi.cs.
2. In the Visual Studio window, click Tools > NuGet Package Manager > Manage NuGet Packages For Solution.
3. On the NuGet Package Manager tab, click Browse.
4. Type **Microsoft asp.net web api owin** and press Enter.
5. Click the Microsoft.AspNet.WebApi.Owin package.
6. On the right side of the NuGet Manager tab, click the check box beside your project.
7. Click the Install button.
8. On the Preview Changes window, click OK.
9. On the License Acceptance, click the I Accept button.
10. Open the Startup.WebApi.cs file and change the content of the file with the content shown in [Listing 3-16](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex16).

**Listing 3-16** Startup.WebApi.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-16a)

// C#. ASP.NET.

using Microsoft.Owin.Security.OAuth;

using Owin;

using System.Web.Http;

namespace <your\_project's\_name>

{

public partial class Startup

{

public void ConfigureWebApi(IAppBuilder app)

{

var config = new HttpConfiguration();

// Web API configuration and services

// Configure Web API to use only bearer token authentication.

config.SuppressDefaultHostAuthentication();

config.Filters.Add(new HostAuthenticationFilter(OAuthDefaults

.AuthenticationType));

// Web API routes

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

app.UseWebApi(config);

}

}

}

1. Open the Startup.cs file and add the following line at the end of the Configuration() method:

ConfigureWebApi(app);

Once you have implemented the resource server in your application, you should be able to make requests to the authorization server to get access to the resource published by the resource server. As you saw in the OAuth workflow, you need to get authenticated by the authorization server before you can get an access token. This means that you need to be logged in to the application before being able to make any requests to the /OAuth/Authorize endpoint.

Now you can create your client application that makes requests to the authorization server and resource server. That client application can be the same application that you used for implementing the authorization and resource servers. You are going to modify the default MVC template for making requests to the Authorization and Resource servers.

1. In the Visual Studio window, click Tools > NuGet Package Manager > Manage NuGet Packages For Solution.
2. In the NuGet Package Manager tab, click Browse.
3. Type **DotNetOpenAuth.OAuth2.Client** and press Enter.
4. Click the DotNetOpenAuth.OAuth2.Client package. This NuGet package eases the interaction with OAuth servers.
5. On the right side of the NuGet Manager tab, click the check box beside your project.
6. Click the Install button.
7. On the Preview Changes window, click OK.
8. Open the ManageController.cs file.
9. Add the following using statements to the ManageController.cs file:
   * **using System;**
   * **using System.Linq;**
   * **using System.Threading.Tasks;**
   * **using System.Web;**
   * **using System.Web.Mvc;**
   * **using Microsoft.AspNet.Identity;**
   * **using Microsoft.AspNet.Identity.Owin;**
   * **using Microsoft.Owin.Security;**
   * **using AuthorizationServer.Models;**
   * **using AuthorizationServer.Constants;**
   * **using DotNetOpenAuth.OAuth2;**
   * **using System.Net.Http;**
10. Replace the Index() method with the code shown in [Listing 3-17](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex17).

**Listing 3-17** Index method in ManageController.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-17a)

**// C#. ASP.NET.**

public async Task<ActionResult> Index(ManageMessageId? message)

{

ViewBag.StatusMessage =

message == ManageMessageId.ChangePasswordSuccess ? "Your password has been

changed."

: message == ManageMessageId.SetPasswordSuccess ? "Your password has been

set."

: message == ManageMessageId.SetTwoFactorSuccess ? "Your two-factor

authentication provider has been set."

: message == ManageMessageId.Error ? "An error has occurred."

: message == ManageMessageId.AddPhoneSuccess ? "Your phone number was

added."

: message == ManageMessageId.RemovePhoneSuccess ? "Your phone number was

removed."

: "";

var userId = User.Identity.GetUserId();

var model = new IndexViewModel

{

HasPassword = HasPassword(),

PhoneNumber = await UserManager.GetPhoneNumberAsync(userId),

TwoFactor = await UserManager.GetTwoFactorEnabledAsync(userId),

Logins = await UserManager.GetLoginsAsync(userId),

BrowserRemembered = await AuthenticationManager.

TwoFactorBrowserRememberedAsync(userId)

};

ViewBag.AccessToken = Request.Form["AccessToken"] ?? "";

ViewBag.RefreshToken = Request.Form["RefreshToken"] ?? "";

ViewBag.Action = "";

ViewBag.ApiResponse = "";

InitializeWebServerClient();

var accessToken = Request.Form["AccessToken"];

if (string.IsNullOrEmpty(accessToken))

{

var authorizationState = \_webServerClient.ProcessUserAuthorization(

Request);

if (authorizationState != null)

{

ViewBag.AccessToken = authorizationState.AccessToken;

ViewBag.RefreshToken = authorizationState.RefreshToken;

ViewBag.Action = Request.Path;

}

}

if (!string.IsNullOrEmpty(Request.Form.Get("submit.Authorize")))

{

var userAuthorization = \_webServerClient.PrepareRequestUserAuthorization(

new[] { "bio", "notes" });

userAuthorization.Send(HttpContext);

Response.End();

}

else if (!string.IsNullOrEmpty(Request.Form.Get("submit.Refresh")))

{

var state = new AuthorizationState

{

AccessToken = Request.Form["AccessToken"],

RefreshToken = Request.Form["RefreshToken"]

};

if (\_webServerClient.RefreshAuthorization(state))

{

ViewBag.AccessToken = state.AccessToken;

ViewBag.RefreshToken = state.RefreshToken;

}

}

else if (!string.IsNullOrEmpty(Request.Form.Get("submit.CallApi")))

{

var resourceServerUri = new Uri(Paths.ResourceServerBaseAddress);

var client = new HttpClient(\_webServerClient.CreateAuthorizingHandler

(accessToken));

var body = client.GetStringAsync(new Uri(resourceServerUri,

Paths.MePath)).Result;

ViewBag.ApiResponse = body;

}

return View(model);

}

1. Add the following property to the ManageController class:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg151-1a)

private WebServerClient \_webServerClient;

1. Add the following helper method to the ManageController class:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg151-2a)

private void InitializeWebServerClient()

{

var authorizationServerUri = new Uri(Paths.AuthorizationServerBaseAddress);

var authorizationServer = new AuthorizationServerDescription

{

AuthorizationEndpoint = new Uri(authorizationServerUri, Paths.AuthorizePath),

TokenEndpoint = new Uri(authorizationServerUri, Paths.TokenPath)

};

\_webServerClient = new WebServerClient(authorizationServer, Clients.Client1.Id,

Clients.Client1.Secret);

}

1. On the Application\_Start() method in the Global.asax.cs file, add the following line:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg151-3a)

AntiForgeryConfig.SuppressXFrameOptionsHeader = true;

1. Add the following using statement to the Global.asax.cs file:

using System.Web.Helpers;

1. In the Solution Explorer window, click Views > Manage > Index.cshtml.
2. Add the code shown in [Listing 3-18](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex18) after the section Two-Factor Authentication in the Index.cshtml file.

**Listing 3-18** Authorization Code Grant section

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-18a)

**// C#. ASP.NET.**

<dt>Authorization Code Grant Client:</dt>

<dd>

<form id="form1" action="@ViewBag.Action" method="POST">

<div>

Access Token<br />

<input id="AccessToken" name="AccessToken" width="604" type="text"

value="@ViewBag.AccessToken" />

<input id="Authorize" name="submit.Authorize" value="Authorize"

type="submit" />

<br />

<br />

Refresh Token<br />

<input id="RefreshToken" name="RefreshToken" width="604"

type="text" value="@ViewBag.RefreshToken" />

<input id="Refresh" name="submit.Refresh" value="Refresh"

type="submit" />

<br />

<br />

<input id="CallApi" name="submit.CallApi" value="Access Protected

Resource API" type="submit" />

</div>

<div>

@ViewBag.ApiResponse

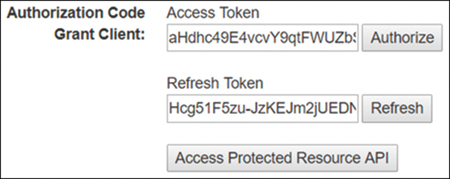
</div>

</form>

</dd>

At this point, your example application is ready for testing the implementation of the different actors that take part in the OAuth workflow. The following steps show how to test your OAuth implementation to ensure that it works correctly:

1. Open the example project in Visual Studio and press F5 to run the project.
2. A new web browser window should open with your web application. Click the Register link located on the top-left corner of the page.
3. On the Register page, add an email address and password and confirm the password. Then click the Register button. You are going to use this user to grant privileges to the OAuth client for making requests to the /OAuth/Me endpoint.
4. Once you have registered the new user, you are automatically logged on and redirected to the Home page.
5. On the Home page, click your user’s email link at the top-left corner of the Home page.
6. On the Manage page, click the Authorize button, which redirects you to the Authorization Server page.
7. On the Authorization Server page, review the information provided and click the Grant button. After you grant access to the OAuth client application, you get the access and refresh token shown in [Figure 3-3](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig03), which is needed to make requests to the resource server.



**Figure 3-3** OAuth Access and Refresh Token

1. Click the Access Protected Resource API to make a request to the /OAuth/Me endpoint. You should get all information stored in the identity claim that you use for making this request, including the scopes bio and notes.

***NEED MORE REVIEW?* OAUTH AUTHORIZATION SERVER**

In this example, you reviewed how to implement the authorization and resource server, the client, and the resource owner on the same web application. Although this is a valid scenario, you usually would find that these roles are implemented on a separate application. The code we reviewed is based on the example explained in the Microsoft Docs article at [*https://docs.microsoft.com/en-us/aspnet/aspnet/overview/owin-and-katana/owin-oauth-20-authorization-server*](https://docs.microsoft.com/en-us/aspnet/aspnet/overview/owin-and-katana/owin-oauth-20-authorization-server). In that article, you can review how to implement each role in separate applications .

**Images EXAM TIP**

When you are working with OAuth2 authentication, remember that you don’t need to store the username and password information in your system. You can delegate that task in specialized authentication servers. Once the user has been authenticated successfully, the authentication server sends an access token that you can use for confirming the identity of the client. This access token needs to be refreshed once the token expires. The OAuth2 can use a refresh token for requesting a new access token without asking the user again for his or her credentials.

**Create and implement shared access signatures**

Until now, all the protection and access control mechanisms that reviewed in this section had to do with protecting the information managed directly by your application. These mechanisms are good if your application manages and presents the information to the user. Still, they are not appropriate for other services that can also store information managed by your application. If your application uses Azure Storage accounts for storing some reports, images, or documents in a table, and you want to grant access to third parties to that information, none of the previously reviewed mechanisms are appropriate for this scenario.

When you are working with storage, you need to control who and how much time a process, person, or application can access your data. Azure Storage allows you to control this access based on several levels of protection:

* **Shared Key Authorization** You use one of the two access keys configured at the Azure Storage account level to construct the correct request for accessing the Azure Storage account resources. You need to use the Authorization Header for using the access key in your request. The access key provides access to the entire Azure Storage account and all its containers, such as blobs, files, queues, and tables. You can consider Azure Storage account keys to be like the root password of the Azure Storage account.
* **Shared Access Signatures** You use Shared Access Signatures (SAS) for narrowing the access to specific containers inside the Storage Account. The advantage of using SAS is that you don’t need to share the Azure Storage account’s access key. You can also configure a higher level of granularity when setting access to your data.

The drawback of using shared access keys is that if either of the two access keys is exposed, the Azure Storage account and all the containers and data in the Azure Storage account are also exposed. The access keys also allow us to create or delete elements in the Azure Storage account.

Shared access signatures provide you with a mechanism for sharing access with clients or applications to your Azure Storage account without exposing the entire account. You can configure each SAS with a different level of access to each of the following:

* **Services** You can configure SAS for granting access only to the services that you require, such as blob, file, queue, or table.
* **Resource types** You can configure access to a service, container, or object. For the Blob service, this means that you can configure the access to API calls at the service level, such as list containers. If you configure the SAS token at the container level, you can make API calls like getting or setting container metadata or creating new blobs. If you decide to configure the access at the object level, you can make API calls like creating or updating blobs in the container.
* **Permissions** Configure the action or actions that the user is allowed to perform in the configured resources and services.
* **Date expiration** You can configure the period for which the configured SAS is valid for accessing the data.
* **IP addresses** You can configure a single IP address or range of IP addresses that are allowed to access your storage.
* **Protocols** You can configure whether the access to your storage is performed using HTTPS-only or HTTP and HTTPS protocols. You cannot grant access to the HTTP-only protocol.

Azure Storage uses the values of previous parameters for constructing the signature that grants access to your storage. You can configure three different types of SAS:

* **User delegation SAS** This type of SAS applies only to Blob Storage. You use an Azure Active Directory user account for securing the SAS token
* **Account SAS** Account SAS controls access to the entire Storage Account. You can also control access to operations at the service level, like getting service stats or getting or setting service properties. You need to use the storage account key for securing this kind of SAS.
* **Service SAS** Service SAS delegates access to only specific services inside the Storage Account. You need to use the storage account key for securing this kind of SAS.

Regardless of the SAS type, you need to construct a SAS token for access. You append this SAS token to the URL that you use for accessing your storage resource. One of the parameters of a SAS token is the signature. The Azure Storage Account service uses this signature to authorize access to the storage resources. The way you create this signature depends on the SAS type that you are using.

For user delegation SAS, you need to use a user delegation key created using Azure Active Directory (Azure AD) credentials. The user used for creating this delegation key needs to have granted the Microsoft.Storage/storageAccounts/blobServices/generateUserDelegationKey/action Role-Base Access Control permission. We review role-based access control authorization in the “[Control access to resources by using role-based access controls (RBAC)](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev2sec4)” later in this chapter.

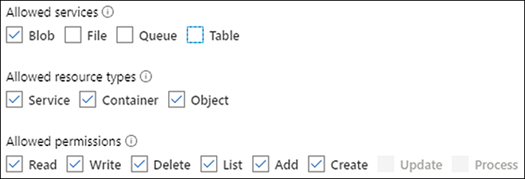
For service or account SAS, you need to use the Azure Storage Account key for creating the signature that you have to include in the SAS token. For constructing the SAS URI for an Account SAS, you need to use the parameters shown in [Table 3-1](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03tab1).

**Table 3-1** Account SAS URI parameters

| **Parameter Name** | **URI Parameter** | **Required** | **Description** |
| --- | --- | --- | --- |
| api-version | api-version | NO | You can set the version of the storage service API that processes your request. |
| SignedVersion | sv | YES | Sets the version of the signed storage service used to authenticate your request. This version should be 2015-04-05 or later. |
| SignedServices | ss | YES | Sets the services to which you grant access. You can grant access to more than one service by combining the allowed values:   * **Blob** You need to use the value (b) in the SAS URI. * **Queue** You need to use the value (q) in the SAS URI. * **Table** You need to use the value (t) in the SAS URI. * **File** You need to use the value (f) in the SAS URI. |
| SignedResourceTypes | Srt | YES | Sets the resource type to which you grant access. You can configure more than one resource type simultaneously by combining more than one of the allowed values:   * **Service** You need to use the value (s) in the SAS URI. * **Container** You need to use the value (c) in the SAS URI. * **Object** You need to use the value (o) in the SAS URI. |
| SignedPermission | sp | YES | Configures the permissions that you grant to the resource types and services configured on previous parameters. Not all permissions apply to all resource types and services. The following list only shows the permissions that apply to the Blob service:   * **Read** You need to use the value (r) in the SAS URI. * **Write** You need to use the value (w) in the SAS URI. * **Delete** You need to use the value (d) in the SAS URI. * **List** You need to use the value (l) in the SAS URI. * **Add** You need to use the value (a) in the SAS URI. * **Create** You need to use the value (c) in the SAS URI.   If you set a permission that is meaningful only for a service or resource type that you didn’t set on the previous parameters, the permission is silently ignored. |
| SignedStart | st | NO | Sets the time and date at which the SAS token is valid. It must be expressed in UTC using ISO 8601 format:   * YYYY-MM-DD * YYYY-MM-DDThh:mmTZD * YYYY-MM-DDThh:mm:ssTZD |
| SignedExpiry | se | YES | Sets the time and date in which the SAS token becomes invalid. It must be expressed in UTC using ISO 8601 format. |
| SignedIP | sip | NO | Sets the IP or range of IP addresses from which the storage service accepts requests. When using ranges of IPs, the limits are included in the range. |
| SignedProtocol | spr | NO | Sets the protocol allowed to request the API. Correct values are   * HTTPS only (https) * HTTP and HTTPS (https, http) |
| Signature | sig | YES | This is an HMAC-SHA256–computed string encoded using Base64 that the API uses for authenticating your request. You calculate the signature based on the parameters that you provided in the SAS URI. This signature must be valid to process your request. |

Use the following procedure for constructing and testing your own account SAS token:

1. Sign in to the management portal ([*http://portal.azure.com*](http://portal.azure.com/)).
2. In the search box at the top of the Azure portal, type the name of your Storage Account.
3. On the Storage Account blade, click Shared Access Signature in the Settings section.
4. On the Shared Access Signature panel, deselect the File, Table, and Queue check boxes under Allowed Services, as shown in [Figure 3-4](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig04). Leave the Blob check box selected.



**Figure 3-4** Configuring the Account SAS policy

1. Ensure that all options in Allowed Resource Types and Allowed Permissions are checked, as shown in [Figure 3-4](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig04).
2. In the Start And Expiry Date/Time section, set a date for a start and ending date and time during which the Azure Storage Account accepts requests using this token.
3. Ensure that Allowed IP addresses have no value in the text box, and HTTPS Only is selected in the Allowed Protocols section.
4. In the Signing Key drop-down menu, make sure that you have selected the Key1 value.
5. Click the Generate SAS And Connection String button at the bottom of the panel.
6. Copy the Blob Service SAS URL. Now you can test your SAS token, using a tool such as Postman, curl, a web browser, or Microsoft Azure Storage Explorer.
7. Open Microsoft Azure Storage Explorer. If you don’t have this tool installed, you can download it from [*https://azure.microsoft.com/en-us/features/storage-explorer/*](https://azure.microsoft.com/en-us/features/storage-explorer/).
8. On the Microsoft Azure Storage Explorer window, on the left side of the window, click the button with a plug icon. This button opens the Connect dialog box.
9. In the Connect dialog box, select the Use A Shared Access Signature (SAS) URI option.
10. Click the Next button on the bottom side of the dialog box.
11. In the Attach With SAS URI, type a name for your connection in the Display Name text box.
12. In the URI text box, paste the URL that you copied in step 10.
13. Click the Next button.
14. Click the Connect button.

Once the connection is created, you should be able to view your Blob Storage service and create new containers or blobs inside the containers.

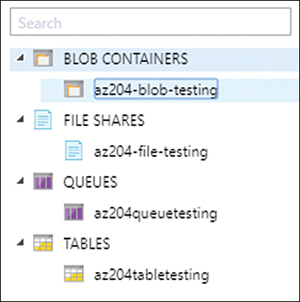
If you need to narrow the access to your resources and limit it only to tables or entities, you can create a Service SAS. This type of SAS token is quite similar to an Account SAS; you need to create a URI that you append to the URL that you use to request your Blob Storage service. Account and Service SAS share most of the URI parameters, although some parameters are specific to the service, and you need to consider them when creating your Service SAS token. [Table 3-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03tab2) shows the parameters that you need to set for creating a Blob Service SAS. Other Azure Storage services require different parameters.

**Table 3-2** BLOB Service SAS URI parameters

| **Parameter Name** | **URI Parameter** | **Required** | **Description** |
| --- | --- | --- | --- |
| SignedVersion | Sv | YES | Sets the version of the signed storage service used to authenticate your request. This version should be 2015-04-05 or later. |
| SignedResource | sr | YES | Sets the type of shared resource:   * **Blob** You need to use the value (b) in the SAS URI. * **Container** You need to use the value (c) in the SAS URI. |
| SignedPermission | Sp | YES | Configures the permissions that you grant to the shared resource. You need to omit this parameter if you decide to use a Stored Access Policy. |
| SignedStart | st | NO | Sets the time and date at which the SAS token is valid. It must be expressed in UTC using ISO 8601 format:   * YYYY-MM-DD * YYYY-MM-DDThh:mmTZD * YYYY-MM-DDThh:mm:ssTZD   If you use an API version 2012-02-12 or later, the difference between signedstart and signedexpiry cannot be greater than one hour unless you are using a container policy. |
| SignedExpiry | se | YES | Sets the time and date in which the SAS token becomes invalid. It must be expressed in UTC using ISO 8601 format.  You need to omit this parameter if you decide to use a Stored Access Policy. |
| SignedIP | sip | NO | Sets the IP or range of IP addresses from which the storage service accepts requests. When using ranges of IPs, the limits are included in the range.  You need to omit this parameter if you decide to use a Stored Access Policy. |
| SignedProtocol | spr | NO | Sets the protocol allowed to request the API. Valid values are   * HTTPS only (https) * HTTP and HTTPS (https, http) |
| SignedIdentifier | Si | NO | Relates the SAS URI that you are constructing with a Stored Access Policy on your Storage Account. Using Stored Access Policies provides a greater level of security. |
| Signature | sig | YES | This is an HMAC-SHA256 computed string encoded using Base64 that the API uses for authenticating your request. You calculate the signature based on the parameters that you provided in the SAS URI. This signature must be valid to process your request. |
| Cache-Control | rscc | NO | Requires version (sv) set to 2013-08-15 or later for Blob service and 2015-02-21 or later for File service. |
| Content-Disposition | rscd | NO | Requires version (sv) set to 2013-08-15 or later for Blob service and 2015-02-21 or later for File service. |
| Content-Encoding | rsce | NO | Requires version (sv) set to 2013-08-15 or later for Blob service and 2015-02-21 or later for File service. |
| Content-Language | rscl | NO | Requires version (sv) set to 2013-08-15 or later for Blob service and 2015-02-21 or later for File service. |
| Content-Type | rsct | NO | Requires version (sv) set to 2013-08-15 or later for Blob service and 2015-02-21 or later for File service. |

The following example shows how to create a Shared Access Signature for a blob container. This SAS token grants access to the blob container and all blobs stored inside the blob container. For this example, you need an Azure Storage Account with a blob container that is configured with the private access level:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. In the search text box at the top of the Azure portal, type the name of your Azure Storage Account.
3. In the Results list, click the name of your Azure Storage Account.
4. On your Azure Storage Account’s blade, click StorageExplorer (preview) in the navigation menu on the left side of the blade.
5. On the Storage Explorer (preview) panel shown in [Figure 3-5](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig05), expand the Blob Containers node and right-click the container, which you need to grant access.



**Figure 3-5** Storage services in the Storage Explorer (preview)

1. In the contextual menu over your blob container, click Get Shared Access Signature.
2. In the Shared Access Signature panel shown in [Figure 3-6](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig06), configure the Start Time, Expiry Time, and Permissions that you want to grant to the SAS token.



**Figure 3-6** Creating a Shared Access Signature

1. Click the Create button at the bottom of the panel.
2. On the Shared Access Signature panel, copy the URL of the newly generated SAS. You can share this SAS URL with any third party who needs to access this specific blob.

You can use these same steps for creating a SAS token for a single blob in a container. Just navigate using the Storage Explorer to the blob that you want to share, right-click the blob, and click Get Shared Access Signature in the contextual menu.

As you can imagine, one drawback of using this approach is that anyone who has access to the SAS URL can access the information protected by that SAS. You can improve the security of the SAS tokens by creating a Stored Access Policy and attaching the policy to the SAS token. Stored Access Policies allows you to define access policies that are associated and stored with the table that you want to protect. When you define a Stored Access Policy, you provide an identifier to the policy. Then you use this identifier when you construct the Service SAS token. You need to include this identifier when you construct the signature that authenticates the token and is part of the SAS itself.

The advantage of using a Stored Access Policy is that you define and control the validity and expiration of the policy without needing to modify the Service SAS token. Using a Stored Access Policy also improves security by hiding the details of the Access Policy from the user, as you just provide the name of the Stored Access Policy. You can associate up to five different stored access policies.

***NEED MORE REVIEW?* WORKING WITH STORED ACCESS POLICIES**

Working with Stored Access Policies is similar to working with ad-hoc access policies. You can review how to work with Stored Access Policies by reviewing the following articles:

* [*https://docs.microsoft.com/en-us/rest/api/storageservices/define-stored-access-policy*](https://docs.microsoft.com/en-us/rest/api/storageservices/define-stored-access-policy)
* [*https://docs.microsoft.com/en-us/azure/storage/common/storage-stored-access-policy-define-dotnet*](https://docs.microsoft.com/en-us/azure/storage/common/storage-stored-access-policy-define-dotnet)

The following example shows how to create a user delegation SAS token using a .NET Core console application:

1. Open Visual Studio Code and create a folder for your project.
2. In the Visual Studio Code Window, open a new terminal.
3. Use the following command to create a new console project:

dotnet new console

1. Use the following command to install NuGet packages:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg161a)

dotnet add package <NuGet\_package\_name>

1. Install the following NuGet packages:
   * Azure.Storage.Blobs
   * Azure.Identity
2. Open the Program.cs file and replace the content with the code shown in [Listing 3-19](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex19).

**Listing 3-19** Program.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-19a)

**// C#. ASP.NET.**

using System;

using Azure.Storage.Blobs;

using Azure.Storage.Blobs.Models;

using Azure.Storage.Sas;

using Azure;

using Azure.Identity;

namespace ch3\_1\_2

{

class Program

{

static void Main(string[] args)

{

string storageAccount = "az204testing";

DateTimeOffset startTimeKey = DateTimeOffset.UtcNow;

DateTimeOffset endTimeKey = DateTimeOffset.UtcNow.AddDays(7);

DateTimeOffset startTimeSAS = startTimeKey;

DateTimeOffset endTimeSAS = startTimeSAS.AddDays(1);

Uri blobEndpointUri = new Uri($"https://{storageAccount}.blob.core.windows.

net");

var defaultCredentials = new DefaultAzureCredential(true);

BlobServiceClient blobClient = new BlobServiceClient(blobEndpointUri,

defaultCredentials);

//Get the key. We are going to use this key for creating the SAS

**UserDelegationKey key = blobClient.GetUserDelegationKey(startTimeKey,**

**endTimeKey);**

System.Console.WriteLine($"User Key Starts on: {key.SignedStartsOn}");

System.Console.WriteLine($"User Key Expires on: {key.SignedExpiresOn}");

System.Console.WriteLine($"User Key Service: {key.SignedService}");

System.Console.WriteLine($"User Key Version: {key.SignedVersion}");

//We need to use the BlobSasBuilder for creating the SAS

**BlobSasBuilder blobSasBuilder = new BlobSasBuilder()**

**{**

**StartsOn = startTimeSAS,**

**ExpiresOn = endTimeSAS**

**};**

//We set the permissions Create, List, Add, Read, and Write

**blobSasBuilder.SetPermissions("clarw");**

**string sasToken = blobSasBuilder.ToSasQueryParameters**

**(key, storageAccount).ToString();**

System.Console.WriteLine($"SAS Token: {sasToken}");

}

}

}

***NOTE* AUTHORIZATIONPERMISSIONMISMATCH**

If you get an exception while running the code in [Listing 3-19](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex19), and the exception message is something similar to “This request is not authorized to perform this operation using this permission” with the error code AuthorizationPermissionMismatch, don’t worry; there is nothing wrong with your code. This exception happens because the user you are using for running this code doesn’t have enough privileges for getting a user delegation key. You can solve this issue by granting the correct permissions to your user. We are going to review how to grant privileges by using role-based access control in a later section .

As you can see in the piece of code in bold in [Listing 3-19](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex19), you use the GetUserDelegationKey() method for getting a user delegation key for your Azure Storage Account. The user you are using for getting this key needs to have assigned the Microsoft.Storage/storageAccounts/blobServices/generateUserDelegationKey/action permission, otherwise you get an exception.

Once you have your user delegation key, you use the BlobSasBuilder class for creating an object that constructs the SAS token for you. Using the instance of the BlobSasBuilder class, you can configure the permissions that you need for accessing the container. In this case, you use the SetPermission() method with the parameter clarw that matches with the permissions shown in [Table 3-1](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03tab1). In this example, because we didn’t set any container name, we get a SAS token for the Azure Blob Storage Account.

Using the ToSasQueryParameters() method from the BlobSasBuilder class, you get the actual SAS token. You need to provide the user delegation key that you obtained previously to this method for getting the SAS token.

Once you get your SAS token, you can use it for accessing your Azure Storage Account. The code in [Listing 3-20](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex20) shows how to interact with your Azure Storage Account, using the SAS token that you created in [Listing 3-19](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex19). If you want to test this code, just replace the content of the Program.cs file that you created in the previous example, with the content in [Listing 3-20](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex20). Before replacing your code, you need to add the System.IO NuGet Package by running the following command in your Visual Studio Code terminal window:

dotnet add package System.IO

**Listing 3-20** Program.cs extension

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-20a)

**// C#. ASP.NET.**

using System;

using Azure.Storage.Blobs;

using Azure.Storage.Blobs.Models;

using Azure.Storage.Sas;

using Azure;

using Azure.Identity;

using System.IO;

namespace ch3\_1\_2

{

class Program

{

static void Main(string[] args)

{

string storageAccount = "az204testing";

string containerName = "az204-blob-testing";

string blobName = System.IO.Path.GetRandomFileName();

DateTimeOffset startTimeKey = DateTimeOffset.UtcNow;

DateTimeOffset endTimeKey = DateTimeOffset.UtcNow.AddDays(7);

DateTimeOffset startTimeSAS = startTimeKey;

DateTimeOffset endTimeSAS = startTimeSAS.AddYears(1);

Uri blobEndpointUri = new Uri($"https://{storageAccount}.blob.core.

windows.net");

var defaultCredentials = new DefaultAzureCredential(true);

BlobServiceClient blobClient = new BlobServiceClient(blobEndpointUri,

defaultCredentials);

//Get the key. We are going to use this key for creating the SAS

UserDelegationKey key = blobClient.GetUserDelegationKey(startTimeKey,

endTimeKey);

Console.WriteLine($"User Key Starts on: {key.SignedStartsOn}");

Console.WriteLine($"User Key Expires on: {key.SignedExpiresOn}");

Console.WriteLine($"User Key Service: {key.SignedService}");

Console.WriteLine($"User Key Version: {key.SignedVersion}");

//We need to use the BlobSasBuilder for creating the SAS

BlobSasBuilder blobSasBuilder = new BlobSasBuilder()

{

BlobContainerName = containerName,

BlobName = blobName,

Resource = "b",

StartsOn = startTimeSAS,

ExpiresOn = endTimeSAS,

Protocol = Azure.Storage.Sas.SasProtocol.Https

};

//We set the permissions Create, List, Add, Read, and Write

blobSasBuilder.SetPermissions(BlobSasPermissions.All);

string sasToken = blobSasBuilder.ToSasQueryParameters

(key, storageAccount).ToString();

Console.WriteLine($"SAS Token: {sasToken}");

//We construct the full URI for accessing the Azure Storage Account

**UriBuilder blobUri = new UriBuilder()**

**{**

**Scheme = "https",**

**Host = $"{storageAccount}.blob.core.windows.net",**

**Path = $"{containerName}/{blobName}",**

**Query = sasToken**

**};**

//We create a random text file

using (System.IO.StreamWriter sw = System.IO.File.CreateText(blobName))

{

sw.Write("This is a testing blob for uploading using user delegated SAS

tokens");

}

**BlobClient testingBlob = new BlobClient(blobUri.Uri);**

**testingBlob.Upload(blobName);**

//Now we download the blob again and print the content.

Console.WriteLine($"Reading content from testing blob {blobName}");

Console.WriteLine();

BlobDownloadInfo downloadInfo = testingBlob.Download();

using (StreamReader sr = new StreamReader(downloadInfo.Content, true))

{

string line;

while ((line = sr.ReadLine()) != null)

{

Console.WriteLine(line);

}

}

Console.WriteLine();

Console.WriteLine("Finished reading content from testing blob");

}

}

}

We have put the essential parts in bold in [Listing 3-20](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex20). When you need to use the SAS token for working Azure Storage accounts, you need to construct the correct SAS token for the element that you are working with. This means, if you are going to work with a container, then you need to create a SAS token for that container and get a reference to the container using the BlobContainerClient. Once you have the reference to the container, you can keep working with other child elements without needing to create a new SAS token for each element inside the container.

In the example in [Listing 3-20](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex20), we create a random text file with some content that we uploaded to the container and then downloaded again. Then we created a SAS token for uploading the random text file. Notice that we created the SAS token pointing to a blob doesn’t even exist. Once we have the correct SAS token, with the correct permissions, we created a BlobClient object using the URI pointing to the final location in the Azure Blob Storage account inside the container. We use the SAS token as the query parameter of the URI. Once we have our BlobClient object representing the blob, we can perform all the needed operations without needing the create a new SAS token for the same blob, as long as the token has not expired.

***NEED MORE REVIEW?* SHARED ACCESS SIGNATURES**

If you want to read more about how to work with Shared Access Signatures, not only with the Azure Blob Storage service, but with other Azure Storage services, like Tables, Queue, or Files, you can review the article at [*https://docs.microsoft.com/en-us/rest/api/storageservices/delegate-access-with-shared-access-signature*](https://docs.microsoft.com/en-us/rest/api/storageservices/delegate-access-with-shared-access-signature).

**Images *EXAM TIP***

If you plan to work with user delegation SAS, you need to consider that this type of SAS is available only for Azure Blob Storage and Azure Data Lake Storage Gen2. You cannot use either Stored Access Policies when working with user delegation SAS.

**Register apps and use Azure Active Directory to authenticate users**

You can secure access to the information managed by your application by using several mechanisms, like form-based authentication, SSL authentication, Windows authentication, or OAuth2 authentication, among others. Each of these mechanisms has advantages and disadvantages.

The “[Implement OAuth2 authentication](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev2sec1)” section earlier in this chapter reviewed how to use OAuth2 authentication with a basic web application. When we reviewed the OAuth2 concepts in that section, you saw that in the OAuth2 authentication flow, there is a security server that takes care of providing the security mechanisms for authenticating the users. Once the security server authenticates, the server emits a token that your application can validate and use for authenticating the request from your application’s client. When working with the security server, you can use your own implementation of an OAuth2 server, or you can rely on third-party security services, like Facebook, Google, or LinkedIn, among others.

Microsoft also provides the ability to use its services for OAuth2 authentication. Microsoft provides OAuth2 authentication through its Azure Active Directory identity service. On its most basic layer, this is a free service that you can use if you want your application’s users to be able to log in using Microsoft Outlook.com accounts for personal accounts or the Azure Active Directory accounts for professional accounts.

Before your application can use the Azure Active Directory service for authenticating your application’s users, you need to register the application in your Azure Active Directory tenant. When you are registering your application, there are some points that you need to consider before proceeding to the registration:

* **Supported account types** You need to consider whether the users of your application would be
  + **Users from your organization only** Any person that have an user account in your Azure Active Directory tenant would be able to use your application.
  + **Users from any organization** You use this option when you want any user with a professional or educational Azure Active Directory account to be able to log into your application.
  + **Users from any organization or Microsoft accounts** You use this option if you want your users to log into your application by using professional, educational, or any of the freely available Microsoft accounts.
* **Platform** The OAuth2 authentication is not limited to web applications. You can also use this type of authentication with mobile platforms, like iOS or Android, or desktop platforms, like macOS, Console, IoT, Windows, or UWP.

The following procedure shows how to register a web application in the Azure Active Directory:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)),
2. On the Search resources, services, and docs text box on the middle-top of the Azure Portal, type **Azure Active Directory**.
3. On the result list, in the Services section, click Azure Active Directory.
4. On the Azure Active Directory page, in the Manage section, click App Registrations.
5. In the App Registrations blade, click the New Registration button on the top-left corner of the panel.
6. In the Register An Application blade, type the name of your application in the Name text box.
7. In the Supported Account Types option control, select Accounts In This Organizational Directory Only.
8. Click the Register button at the bottom-left corner of the blade.

The previous procedure shows how to make a simple app registration. Now you need to configure your app registration according to your app needs. One of the most critical sets of settings that you need to configure correctly is the Authentication settings, shown in [Figure 3-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig07). You use the Authentication settings for managing the authentication options for your application. In this case, you configure the redirect URLs used by Azure Active Directory for authenticating your application’s requests. If the redirect URL provided by your application doesn’t match with any of the URLs configured in this section, the authentication fails.



**Figure 3-7** Authentication settings

The other two critical sets of settings that you need to consider are Certificates & Secrets and API Permissions. Certificates & Secrets enables you to manage the Certificates and the Secrets that your application needs to use to provide the application’s identity when requesting a token. You use the API Permissions for configuring the needed permission for calling other APIs, either from Microsoft, your organization, or other third-party APIs. The following example shows how to create a simple web application that uses Azure Active Directory authentication. Although you could register the app for this example directly from the wizard in Visual Studio 2019, we prefer to show you how to make an app registration directly from the Azure portal. In this example, you are going to use the app that you registered in the previous procedure. If you didn’t follow that procedure, you should review and follow it before you can proceed with the following example:

1. Open Visual Studio 2019.
2. In the welcome window of Visual Studio 2019, on the Get Started column, click Create A New Project.
3. On the Create A New Project window, on All Languages drop-down menu, select C#.
4. In the Search For Templates text box, type **asp.net**.
5. In the result list, click ASP.NET Web Application (.NET Framework).
6. Click the Next button at the bottom-right corner of the window.
7. On the Configure Your New Project window, type a Project Name, a Location, and a Solution Name for your project.
8. Click the Create button at the bottom-right corner of the window.
9. In the Create A New ASP.NET Web Application window, select the MVC template on the template list in the middle of the left side of the window. MVC is for Model-View-Controller.
10. On the right side of the Create A New ASP.NET Web Application window, on the Authentication section, ensure the Authentication is set to No Authentication.
11. Click the Create button at the bottom-right corner of the window.
12. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)) and navigate to the app that you registered in the previous example.
13. In the Overview blade of your app in the Azure portal, copy the value of the parameter Application (client) ID. You need this value for a later step.
14. In the Manage section on the left side of your App blade, click Certificates & Secrets.
15. On the Certificates & Secrets blade, in the Client Secrets area, click the New Client Secret button.
16. Type a description in the text box for this client secret.
17. Click the Add button.
18. In the Client Secrets area, on the list of client secrets, copy the value of the client secret that you just created. You need this value in a later step.
19. On the Solution Explorer window in your Visual Studio 2019 window, right-click the Connected Services node.
20. Click Add Connected Service in the contextual menu.
21. In the Connected Services Windows, click Authentication with Azure Active Directory.
22. In the Configure Azure AD Authentication window, in the Introduction section, click the Next button at the bottom right of the window.
23. In the Single Sign-On section, in the Domain drop-down menu, type the name of your tenant. You can find this information in the Azure portal, in the Overview blade of your Azure Active Directory tenant.
24. On the Configuration Settings area, select Use Settings From An Existing Azure AD Application To Configure Your Project.
25. In the Client ID text box, paste the value that you copied in step 13.
26. Leave Redirect URI blank.
27. Click the Next button at the bottom right of the window.
28. In the Directory Access section, check the Read Directory Data option.
29. In the Client Secret text box, copy the client secret that you created in step 18.
30. Click the Finish button.
31. In the Azure AD Authentication window, wait for the wizard to add all the needed code to your application.
32. In the Solution Explorer, click the name of your project and press F4.
33. In the Properties window, copy the value of the SSL URL setting. You need this value in a later step.
34. Open the Azure portal and navigate to your registered app in your Azure Active Directory tenant.
35. On your registered app blade, in the Manage section, click Authentication.
36. On the Authentication blade, in the Platform Configurations area, click the Add A Platform button.
37. On the Configure Platforms panel, in the Web Applications section, click Web.
38. On the Redirect URI text box, copy the value of the SSL URL setting that you copied in step 33.
39. In the Implicit Grant section, check ID Tokens.
40. Click the Configure button.
41. In your Visual Studio 2019 window, press F5 for running your project.

Once you execute the web application, a generic Microsoft login page should appear in your browser. You need to provide a valid user account from your tenant to log in. At this point, your application uses the Azure Active Directory for authenticating users. Additionally, you can also read information from your tenant.

When you run the Authentication with the Azure Active Directory wizard, there are some code changes that you should understand. The wizard adds some properties to the appSettings section in the web.config file. These properties represent relevant settings needed for connecting to your Azure Active Directory tenant:

* **ida:ClientId** This is the ID representing your registered application in your Azure Active Directory tenant.
* **ida:AADInstance** Provides the instance that you are going to use for authentication. In most situations, you use general public instances. You only need to change this value if your tenant is hosted in isolated instances like Government, China, or Germany.
* **ida:Domain** This is your Azure Active Directory tenant, where you registered your app.
* **ida:TenantId** This is the ID representing the tenant where you registered your app.
* **ida:PostLogoutRedirectUri** This is the URL where Microsoft redirects the user once the authentication process finished successfully. This value must match with the value configured in your app registration in the Azure portal.
* **ida:ClientSecret** The client secret is similar to the password that your application uses for authenticating against the Azure Active Directory service before it can interact with the APIs protected by the identity service.

As with any other authorization system in C#, you need to add the [Authorized] attribute to any resource that you want to protect. In this case, the wizard adds this attribute to any existing controller in your application. Finally, the most critical piece of code is the one used for configuring the OpenID/OAuth2 authentication. [Listing 3-21](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex21) shows the code snippet added to your Startup.Auth.cs file for connecting your application with Azure Active Directory.

**Listing 3-21** Program.cs extension

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-21a)

**// C#. ASP.NET.**

app.UseOpenIdConnectAuthentication(

new OpenIdConnectAuthenticationOptions

{

ClientId = clientId,

Authority = Authority,

PostLogoutRedirectUri = postLogoutRedirectUri,

Notifications = new OpenIdConnectAuthenticationNotifications()

{

// If there is a code in the OpenID Connect response, redeem //

it for an access token and refresh token, and store those

// away.

AuthorizationCodeReceived = (context) =>

{

var code = context.Code;

ClientCredential credential = new ClientCredential

(clientId, appKey);

string signedInUserID = context.AuthenticationTicket

.Identity.FindFirst(ClaimTypes

.NameIdentifier).Value;

AuthenticationContext authContext =

new AuthenticationContext(Authority,

new ADALTokenCache(signedInUserID));

return authContext.AcquireTokenByAuthorizationCodeAsync(

code, new Uri(HttpContext.Current.Request.Url

.GetLeftPart(UriPartial.Path)), credential,

graphResourceId);

}

}

});

***NEED MORE REVIEW?* AUTHENTICATION WITH AZURE AD WIZARD**

If you want to read more about the changes made by the Authentication with Azure AD wizard, review the article at [*https://docs.microsoft.com/en-us/azure/active-directory/develop/vs-active-directory-dotnet-what-happened*](https://docs.microsoft.com/en-us/azure/active-directory/develop/vs-active-directory-dotnet-what-happened).

**Images *EXAM TIP***

When you are registering a new application in your Azure Active Directory tenant, you need to consider which will be your target user. If you need for any user from any Azure Active Directory organization to be able to log into your application, you need to configure a multitenant app. In those multitenant scenarios, the app registration and management is always performed in your tenant and not in any other external tenant.

**Control access to resources by using role-based access controls (RBAC)**

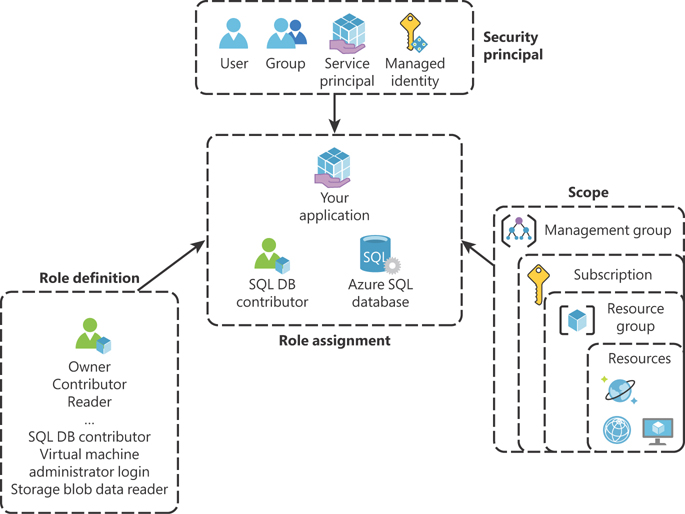
When you are working with your Azure subscription, there are situations where you need to grant access to other users. These users may need access only to specific resources inside your subscription, like a specific resource group or to an Azure SQL Database. You can achieve this level of granularity by using role-based access controls (RBAC).

Built on top of the Azure Resource Manager, RBAC provides fine-grained access control to the different resources in an Azure subscription. When working with RBAC, you need to consider the following concepts:

* **Security principal** This is the entity that requests permission for doing an action. A security principal can be one of the following:
  + **User** This is an individual who has a profile in an Azure Active Directory tenant. You are not limited to your own tenant. You can assign a role to users in other tenants as well.
  + **Group** This is a set of users.
  + **Service principal** This is like a user for an application. A service principal represents an application inside the tenant.
  + **Managed identity** This kind of identity represents cloud applications that need to access resources in your Azure tenant. Azure automatically manages this kind of identity.
* **Permission** This is the action that you can perform with a resource. An example of an action would be requesting a user delegation key for creating a SAS token. Another example of an action is listing the content of a container. You cannot directly assign a role to a security principal. You always need to use a role or role definition.
* **Role definition** Usually known as just role for short, a role definition is a collection of permissions. You assign a role to a security principal. There are a lot of predefined roles in Azure that you can use for managing access to the resources. There are four fundamental roles:
  + **Owner** Grants full access to all resources in the scope.
  + **Contributor** Grants modify access to all resources in the scope. You can perform all modification operations, including deleting, with the resources in the scope. You cannot grant roles to other security principals.
  + **Reader** Grants read access to all resources in the scope.
  + **User Access Administrator** Useful only for managing user access to Azure resources.

Aside from these four fundamental built-in roles, there are roles specific to each service, like Virtual Machine Contributor or Cosmos DB Account Reader.

* **Scope** This is the group of resources where you assign the role. You can set a role at four different levels: management group (a group of subscriptions), subscription, resource group, and resource. These four levels are organized in a parent-child relationship where the management group is the highest level and resource is the lowest. When you assign a role to level, those permission are inherited by the lower levels. That means that if you grant the Owner role to a user at the subscription level, that user has the Owner privileges in all the resource groups and resources in that subscription.
* **Role assignment** This is the junction between the different pieces of RBAC. A role assignment connects a security principal with a role and a scope. [Figure 3-8](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig08) shows the relationship between the different RBAC elements.



**Figure 3-8** Role-based access control

The following procedure shows how to grant the Contributor role to a resource group:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. In the Search Resources, Services, And Docs text box, type the name of the resource group where you want to grant the Contributor role.
3. In the result list, click the name of the resource group.
4. On the Resource Group blade, click Access Control (IAM) on the left side of the control.
5. On the Access Control (IAM) blade, click the Add button on the top-left corner of the blade.
6. On the contextual menu that appears below the Add button, click Add Role Assignment.
7. On the Add Role Assignment blade, on the Role drop-down menu, select Contributor.
8. Leave the Assign Access To drop-down menu set to the default value.
9. In the Select text box, type the name of the user, group, or service principal that you want to assign the Contributor role.
10. On the result list below the Select text box, click the security principal that you want to assign the role.
11. Click the Save button on the bottom-left corner of the Add Role Assignment blade.

Making role assignments to other levels, like management groups, subscriptions, or resources, is similar to the previous procedure. In general, you make the RBAC role association in the Access Control (IAM) section of each level.

***NEED MORE REVIEW?* RBAC CUSTOM ROLES**

Although Azure provides a good amount of built-in roles, there could be situations where the built-in roles don’t have the right privileges for your needs. In those situations, you can create a custom role. When you define a custom role, you use the role in the same way as you use a built-in role. You need to create a role definition for your role by configuring the needed permissions for your custom role. You can find more information about how to create custom roles by reviewing the article at [*https://docs.microsoft.com/en-us/azure/role-based-access-control/custom-roles*](https://docs.microsoft.com/en-us/azure/role-based-access-control/custom-roles).

**Images *EXAM TIP***

When you are assigning specific service roles, carefully review the permissions granted by the role. In general, granting access to a resource doesn’t grant access to the data managed by that resource. For example, the Storage Account Contributor grants access for managing Storage Accounts but doesn’t grant access to the data itself.

**Skill 3.2: Implement secure cloud solutions**

The previous skill reviewed how to protect access to the data by authenticating and authorizing users who try to access the information managed in your application. This protection is only a portion of the mechanisms that you should put in place for protecting your data. You also need to ensure that all the configuration needed for running your application in the different environments is managed securely. The reason for also securing that configuration is that configuration has the needed passwords, certificates, and secrets for accessing the information managed by your application.

When you encrypt your data, you need to use encryption and decryption keys or secrets for accessing and protecting the data. Storing these secrets and encryption keys is as important as encrypting the data. Losing an encryption or decryption key is similar to losing your house’s keys. The Azure Key Vault allows you to securely store these encryption/decryption keys as well as other secrets or certificates that your applications may require in a secured encryption store in Azure. In conjunction with Managed Identities, the Azure Key Vault services allow you to securely store your secrets without needing to store a password, certificate, or any kind of credentials for accessing your secrets.

**This skill covers how to**

* [Secure app configuration data by using the App Configuration and KeyVault API](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev2sec5)
* [Manage keys, secrets, and certificates by using the KeyVault API](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev2sec6)
* [Implement Managed Identities for Azure resource](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03lev2sec7)

**Secure app configuration data by using the App Configuration and KeyVault API**

Most of today’s medium to large applications are based on distributed architectures. Independently of whether the infrastructure that executes your application is based on virtual machines, containers, serverless computing, or any other type of computing, you need to share the configuration between the elements that execute the same component of your application. For example, if your application runs on an Internet Information Services cluster behind a load balancer, all the virtual machines hosting the IIS service share the same configuration for running your application.

This kind of scenario is where Azure App Configuration becomes a handy tool. Azure App Configuration allows you to store all the necessary configuration for your cloud application in a single repository. Other Azure services also allow you to manage configuration for your apps, but they have some crucial differences that you should consider:

* **Azure App Service settings** As you know, you can create settings for your Azure App Service. These settings apply to the instance that you are configuring. You can even create different settings values for different deployment slots. On the other hand, the Azure App Configuration service is a centralized configuration service that allows you to share the same configuration between different Azure App Service instances. You also need to consider that the Azure App Configuration service is not limited to Azure App Service. You can also use it with containerized applications or with applications running inside virtual machines.
* **Azure Key Vault** Azure Key Vault allows you to securely store passwords, secrets, and any other setting that your application may need. The encryption is performed using hardware-level encryption, among other interesting features like certificates rotation or granular access policies. Although Azure App Configuration encrypts the value of your configuration, Azure Key Vault still provides higher levels of security. You can use the Azure Key Vault in conjunction with Azure App Configuration by creating references in Azure App Configuration to items stored in the Azure Key Vault.

When you work with Azure App Configuration, you need to deal with two different components: the App Configuration store and the SDK. The Azure App Configuration store is the place where you store your configuration. When you configure your Azure App Configuration store, you can choose between two different pricing tiers: Free and Standard. The main difference between the two pricing tiers is the number of stores that you can create in a subscription. In the Free tier, you are limited to one store per subscription whereas you don’t have such a limitation in the Standard tier. Other differences between the two tiers are the maximum size of the store: 10 MB in the Free tier versus 1 GB in the Standard tier, or the size of the key history: 7 days in the Free tier versus 30 days in the Standard tier. You can switch from the Free to the Standard tier at any moment, but you cannot switch back to the Free tier from the Standard tier. The following procedure shows how to create an Azure App Configuration:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Click the Create A Resource button in the Azure Services section at the top of the Azure portal.
3. In the New blade, type **app configuration** in the Search The Marketplace text box.
4. In the result list below the text box, click App Configuration.
5. In the App Configuration blade, click the Create button.
6. In the App Configuration blade, type a name for the App Configuration store in the Resource Name text box. The name must contain only alphanumeric ASCII characters or the hyphen (-) character, and it must be between 5 and 50 characters.
7. Select the subscription where you want to deploy your App Configuration store using the Subscription drop-down menu.
8. In the Resource Group drop-down menu, select the resource group where you want to deploy your App Configuration store. Alternatively, you can create a new resource group by clicking the Create New link below the Resource Group drop-down menu.
9. Select a location in the Location drop-down menu.
10. In the Pricing Tier drop-down menu, select Free.
11. Click the Create button at the bottom of the blade.

Once you have created your Azure App Configuration store, you can create the key-value pairs for storing your configuration. Before you create your first key-value pair, you should review how keys work inside the App Configuration store.

A key is an identifier associated with a value stored in the App Configuration store. You use the key for retrieving a value from the store. Keys are case-sensitive, so “appSample204” and “APPSAMPLE204” are different keys. This is important because some languages or frameworks are case-insensitive for settings, so you should not use case-sensitivity for differencing keys. When naming a key, you can use any Unicode character, except the asterisk (\*), the comma (,), and the back slash (\). If you need to include any of these reserved characters, you need to prepend the back slash (\) escape character. As a best practice, you should consider using namespace when naming your keys. By using a separator character between the different levels, you can create a hierarchy of settings inside your store. As the Azure App Configuration service doesn’t analyze or parse your keys, it is entirely up to you to choose the namespace that better fits your needs. Some examples of keys using namespaces are

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg177-1a)

AppSample:Devel:DbConnection

AppSample:AUS:WelcomeMessage

You can also add a label attribute to a key. By default, the label attribute is null. You can use the label for making values different using the same key. This is especially useful when used for deployment environments: The following three examples are different keys because the labels are different:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg177-2a)

Key = AppSample:DBConnection – Label = Develop

Key = AppSample:DBConnection – Label = Stage

Key = AppSample:DBConnection – Label = Production

When creating a new key-value pair, you have a limit of 10,000 for the size of the pair. This limit applies to the size of the key, plus the size of the optional label, plus the size of the value. You should also bear in mind that the same limitations that apply to the string that you use for the key are the same for value. That is, you can use any Unicode character for the value, except asterisk (\*), comma (,), and back slash (\). If you need to include any of these reserved characters, you need to prepend the back slash (\) escape character.

***NEED MORE REVIEW?* FEATURE MANAGEMENT AND DYNAMIC CONFIGURATION**

You can take advantage of Azure App Configuration for implementing more advanced features like feature management or dynamic configuration. The following articles give you more insight about these features:

* **Feature Management** [*https://docs.microsoft.com/en-us/azure/azure-app-configuration/quickstart-feature-flag-aspnet-core*](https://docs.microsoft.com/en-us/azure/azure-app-configuration/quickstart-feature-flag-aspnet-core)
* **Enable Dynamic configuration** [*https://docs.microsoft.com/en-us/azure/azure-app-configuration/enable-dynamic-configuration-aspnet-core*](https://docs.microsoft.com/en-us/azure/azure-app-configuration/enable-dynamic-configuration-aspnet-core)

After you’ve reviewed the basics of the Azure App Configuration, you can review how to use this service in your code. The following example is based on the code in [Chapter 2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch02.xhtml#ch02) in the section “[Interact with data using the appropriate SDK](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch02.xhtml#ch02lev2sec3).” In this example, you are going to modify the code for using an Azure App Container store instead of using an AppSettings JSON file:

1. Open Visual Studio Code and create a folder for your project.
2. In the Visual Studio Code Window, open a new terminal.
3. Use the following command to create a new console project:

dotnet new console

1. Use the following command to install NuGet packages:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg178a)

dotnet add package <NuGet\_package\_name>

1. Install the following NuGet packages:
   * Azure.Storage.Blobs
   * Azure.Storage.Common
   * Azure.Identity
   * Microsoft.Extensions.Configuration
   * Microsoft.Extensions.Configuration.AzureAppConfiguration
2. Create a C# class file and name it **AppSettings.cs**.
3. Replace the contents of the AppSettings.cs file with the contents of [Listing 3-22](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex22). Change the name of the namespace to match your project’s name.
4. Create a C# class file and name it **Common.cs**.
5. Replace the contents of the Common.cs file with the contents of [Listing 3-23](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex23).
6. Change the name of the namespace to match your project’s name.
7. Replace the contents of the **Program.cs** file with the contents of [Listing 3-24](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex24). Change the name of the namespace to match your project’s name.

**Listing 3-22** AppSettings.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-22a)

**// C#. ASP.NET.**

using System;

using Microsoft.Extensions.Configuration;

namespace ch3\_2\_1

{

public class AppSettings

{

public string SourceSASConnectionString { get; set; }

public string SourceAccountName { get; set; }

public string SourceContainerName { get; set; }

public string DestinationSASConnectionString { get; set; }

public string DestinationAccountName { get; set; }

public string DestinationContainerName { get; set; }

public static AppSettings LoadAppSettings()

{

var builder = new ConfigurationBuilder();

builder.AddAzureAppConfiguration(Environment.GetEnvironmentVariable

("ConnectionString"));

var config = builder.Build();

AppSettings appSettings = new AppSettings();

appSettings.SourceSASConnectionString = config["TestAZ204:StorageAccount:

Source:ConnectionString"];

appSettings.SourceAccountName = config["TestAZ204:StorageAccount:Source:

AccountName"];

appSettings.SourceContainerName = config["TestAZ204:StorageAccount:

Source:ContainerName"];

appSettings.DestinationSASConnectionString = config["TestAZ204:Storage

Account:Destination:

ConnectionString"];

appSettings.DestinationAccountName = config["TestAZ204:StorageAccount:

Destination:AccountName"];

appSettings.DestinationContainerName = config["TestAZ204:StorageAccount:

Destination:ContainerName"];

return appSettings;

}

}

}

**Listing 3-23** Common.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-23a)

**// C#. ASP.NET.**

using Azure.Storage.Blobs;

namespace ch3\_2\_1

{

public class Common

{

public static BlobServiceClient CreateBlobClientStorageFromSAS(

string SASConnectionString)

{

BlobServiceClient blobClient;

try

{

blobClient = new BlobServiceClient(SASConnectionString);

}

catch (System.Exception)

{

throw;

}

return blobClient;

}

}

}

**Listing 3-24** Program.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-24a)

**// C#. ASP.NET.**

using System.Threading.Tasks;

using System;

using Azure.Storage.Blobs;

namespace ch3\_2\_1

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Copy items between Containers Demo!");

Task.Run(async () => await StartContainersDemo()).Wait();

}

public static async Task StartContainersDemo()

{

string sourceBlobFileName = "Testing.zip";

AppSettings appSettings = AppSettings.LoadAppSettings();

//Get a cloud client for the source Storage Account

BlobServiceClient sourceClient = Common.CreateBlobClientStorageFromSAS(

appSettings.SourceSASConnectionString);

//Get a reference for each container

var sourceContainerReference = sourceClient.GetBlobContainerClient(app

Settings.SourceContainerName);

var destinationContainerReference = sourceClient.GetBlobContainerClient(

appSettings.DestinationContainerName);

//Get a reference for the source blob

var sourceBlobReference = sourceContainerReference.GetBlobClient(

sourceBlobFileName);

var destinationBlobReference = destinationContainerReference.

GetBlobClient(sourceBlobFileName);

//Move the blob from the source container to the destination container

await destinationBlobReference.StartCopyFromUriAsync(sourceBlobReference

.Uri);

}

}

}

[Listings 3-23](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex23) and [3-24](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex24) are mostly the same files that you can find in the example in [Chapter 2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch02.xhtml#ch02) in the section “[Interact with data using the appropriate SDK](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch02.xhtml#ch02lev2sec9).” The file AppSettings.cs shown in [Listing 3-22](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex22) contains all the magic for working with the App Configuration service. As with any regular .NET Core application, you need to create a ConfigurationBuilder object for managing the configuration of the application. Once you get your builder, you use the AddAzureAppConfiguration() extension method for connecting to the App Configuration store. Finally, you use the Build() method from the builder object for loading all the settings stored in the App Configuration store. Once you loaded all the settings, you can access each key-value pair by just using the correct key, as you can see in [Listing 3-24](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex24).

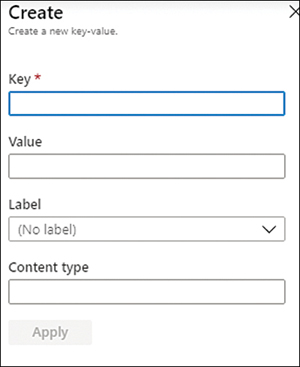
At this point, if you try to run this example, you will get some exceptions because you are not providing the connection string needed for accessing your Azure App Configuration store. You neither defined any key-value pair in your App Configuration store, so even if you were able to access the store, you would get null values. Use the following steps for getting the connection string needed for accessing the App Configuration store and define each of the needed key-value pairs:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. In the Search Resources, Services, and Docs text box at the top of the Azure portal, type the name of your App Configuration store.
3. In your App Configuration Store blade, click Access Keys in the Settings section.
4. In the Access Keys blade, copy one of the connection strings by clicking the blue icon on the right side next to the Connection String text box.
5. In your Visual Studio Code window, open a new terminal and type the following command. Replace the text <*your\_connection\_string*> with the value that you copied in step 4:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg181a)

setx ConnectionString "<*your\_connection\_string*>"

1. Restart your Visual Studio Code window. You need to perform this step to ensure that the environment variable that you defined in the previous step is available for your code.
2. In the Azure portal, in your App Configuration store blade, click Configuration Explorer in the Operations section on the left side of the blade.
3. In the Configuration Explorer, click the Create button at the top-left corner of the blade.
4. In the Create panel, shown in [Figure 3-9](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig09), type the name of the key in the Key text box. Use one of the keys shown in [Listing 3-22](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex22).



**Figure 3-9** Create a new key-value

1. In the Value text box, provide the value for the corresponding key. Remember that you are using values from the example in [Chapter 2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch02.xhtml#ch02) in the section “[Interact with data using the appropriate SDK](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch02.xhtml#ch02lev2sec9).” The correct values are specific to your scenario, but you can use [Listing 2-12](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch02.xhtml#ch02ex12) for your reference.
2. Click the Apply button.
3. Repeat steps 8 to 10 until you create a key-value pair for each setting in [Listing 3-22](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex22). Here is the complete list for your reference:
   * TestAZ204:StorageAccount:Source:ConnectionString
   * TestAZ204:StorageAccount:Source:AccountName
   * TestAZ204:StorageAccount:Source:ContainerName
   * TestAZ204:StorageAccount:Destination:ConnectionString
   * TestAZ204:StorageAccount:Destination:AccountName
   * TestAZ204:StorageAccount:Destination:ContainerName
4. In your Visual Studio Code window, press F5 for running your project. At this point, your code should be able to connect to your Azure App Configuration store and retrieve all the needed settings.

This example shows you the basics for working with Azure App Configuration but also shows some drawbacks that you should consider for production environments. In this example, you defined an environment variable for storing the connection string for connecting to the App Configuration store. Although this could be a valid configuration for testing or developing environments, there are security implications that you should consider for production environments. You should consider using Managed Service Identity for production environments, instead of using connection strings.

Another security improvement that you should consider is storing connection strings directly as a key-value in your App Configuration store. For this kind of sensitive information, you should store it as a secret in an Azure Key Vault and create an Azure Key Vault Reference in your App Configuration store pointing to the right secret. For the sake of brevity, we didn’t include the procedure of how to create Key Vault references, but you can review a complete reference in the article at [*https://docs.microsoft.com/en-us/azure/azure-app-configuration/use-key-vault-references-dotnet-core*](https://docs.microsoft.com/en-us/azure/azure-app-configuration/use-key-vault-references-dotnet-core).

***NEED MORE REVIEW?* BEST PRACTICES**

You can review some best practices when working with Azure App Configuration by reading the article at [*https://docs.microsoft.com/en-us/azure/azure-app-configuration/howto-best-practices*](https://docs.microsoft.com/en-us/azure/azure-app-configuration/howto-best-practices).

**Images *EXAM TIP***

When you are defining your key-value pair, remember that you are limited to a maximum length of 10,000. Remember also that keys are case-sensitive, so “AppSetting” and “appsetting” are treated as different keys.

**Manage keys, secrets, and certificates by using the KeyVault API**

Azure Key Vault is the service provided by Microsoft for securely storing secret keys and certificates in a centralized, secure store. By using Azure Key Vault, your developers no longer need to store this sensitive information on their computers while they are developing an application. Thanks to the identity-based access control, you only need to configure a policy for granting access to the needed service or user principals to the secure store. Another advantage is that you can apply fine-grained access control, allowing access to specific secrets only to the needed application or user.

The next example shows how to use the KeyVault API for creating, reading, updating, or deleting the different elements that you can store in the Azure Key Vault. You need an empty Azure App Service and an Azure Key Vault configured in your Azure subscription to run this example.

1. Open the Azure portal at [*https://portal.azure.com*](https://portal.azure.com/).
2. In the search text box on top of the Azure portal, type the name of your Azure Web App.
3. Click the name of your Azure Web App in the result list below the text box.
4. On the Azure Web App Service blade, click the Identity menu item in the Settings section.
5. In the Status switch control, click the On option.
6. Click Save.
7. In the Enable System Assigned Managed Identity dialog box, click Yes.
8. Once you enable the system-assigned managed identity, you get the Principal or Object ID associated with your Azure App Service.
9. In the search text box at the top of the Azure portal, type the name of your Azure Key Vault. Click the name of your Azure Key Vault in the result list.
10. On the Key Vault blade, click Access Policies in the Settings section in the navigation menu.
11. On the Access Policies blade, click the Add Access Policy link.
12. On the Add Access Policy panel, click the Configure From Template drop-down menu and select the Key, Secret, and Certificate Management option.
13. Click Select Principal.
14. On the Principal panel, type the name of your Azure App Service in the Select text box.
15. In the results list, click the name of your Azure App Service.
16. Click the Select button.
17. On the Add Access Policy panel, click the Add button.
18. On the Access Policies blade, click the Save button in the top-left corner of the blade.
19. Repeat steps 10 through 18 and add the user account that you use for accessing your Azure subscription. You need to add this policy to be able to debug your code using Visual Studio. You need to ensure that you add the policy for granting access to the same user account that you use accessing your Azure subscription from Visual Studio.
20. Open Visual Studio 2019.
21. In the welcome window of Visual Studio 2019, on the Get Started column, click Create A New Project.
22. On the Create A New Project window, on the All Languages drop-down menu, select C#.
23. On the Search For Templates text box type **asp.net**.
24. On the result list, click ASP.NET Web Application (.NET Framework).
25. Click the Next button at the bottom right of the window.
26. On the Configure Your New Project, type a Project Name, a Location, and a Solution Name for your project.
27. Click the Create button at the bottom right of the window.
28. On the Create A New ASP.NET Web Application window, select the MVC template on the template list in the middle of the left side of the window. MVC is for Model-View-Controller.
29. On the right side of the Create A New ASP.NET Web Application window, on the Authentication section, ensure the Authentication is set to No Authentication.
30. Click the Create button at the bottom-right corner of the window.
31. In the Visual Studio window, click Tools > NuGet Package Manager > Manage NuGet Packages For Solution.
32. On the NuGet Package Manager tab, click Browse.
33. Type **Microsoft.Azure.Services.AppAuthentication** and press Enter.
34. Click the Microsoft.Azure.Services.AppAuthentication package.
35. On the right side of the NuGet Manager tab, click the check box next to your project.
36. Click the Install button.
37. In the Preview Changes window, click OK.
38. In the License Acceptance window, click the I Accept button.
39. Repeat steps 32 through 38 and install the Microsoft.Azure.KeyVault package.
40. Open the HomeController.cs file in the Controllers folder.
41. Replace the content of the Index() method with the content of [Listing 3-25](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex25). You may need to add the following namespaces to the HomeController.cs file:
    * Microsoft.Azure.KeyVault
    * Microsoft.Azure.KeyVault.Models
    * Microsoft.Azure.Services.AppAuthentication
    * System.Threading
    * System.Threading.Tasks

**Listing 3-25** Creating, deleting, updating, and reading Key Vault items

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-25a)

**// C#. ASP.NET.**

public ActionResult Index()

{

string keyVaultName = "<YOUR\_VAULT's\_NAME>";

string vaultBaseURL = $"https://{keyVaultName}.vault.azure.net";

//Get a token for accessing the Key Vault.

var azureServiceTokenProvider = new AzureServiceTokenProvider();

//Create a Key Vault client for accessing the items in the vault;

var keyVault = new KeyVaultClient(new KeyVaultClient.AuthenticationCallback

(azureServiceTokenProvider.KeyVaultTokenCallback));

// Manage secrets in the Key Vault.

// Create a new secret

string secretName = "secret-az204";

Task.Run(async () => await keyVault.SetSecretAsync(vaultBaseURL,

secretName,

"This is a secret testing value")).Wait();

var secret = Task.Run(async () => await keyVault.GetSecretAsync

($"{vaultBaseURL}/secrets/{secretName}")).GetAwaiter().GetResult();

// Update an existing secret

Task.Run(async () => await keyVault.SetSecretAsync(vaultBaseURL,

secretName,

"Updated the secret testing value")).Wait();

secret = Task.Run(async () => await keyVault.GetSecretAsync

($"{vaultBaseURL}/secrets/{secretName}")).GetAwaiter().GetResult();

// Delete the secret

Task.Run(async () => await keyVault.DeleteSecretAsync(vaultBaseURL,

secretName)).Wait();

// Manage certificates in the Key Vault

string certName = "cert-az204";

// Create a new self-signed certificate

var policy = new CertificatePolicy

{

IssuerParameters = new IssuerParameters

{

Name = "Self",

},

KeyProperties = new KeyProperties

{

Exportable = true,

KeySize = 2048,

KeyType = "RSA"

},

SecretProperties = new SecretProperties

{

ContentType = "application/x-pkcs12"

},

X509CertificateProperties = new X509CertificateProperties

{

Subject = "CN=AZ204KEYVAULTDEMO"

}

};

Task.Run(async () => await keyVault.CreateCertificateAsync(vaultBaseURL,

certName, policy, new CertificateAttributes { Enabled = true })).Wait();

// When you create a new certificate in the Key Vault it takes some time

// before it's ready.

// We added some wait time here for the sake of simplicity.

Thread.Sleep(10000);

var certificate = Task.Run(async () => await keyVault.GetCertificateAsync

(vaultBaseURL, certName)).GetAwaiter().GetResult();

// Update properties associated with the certificate.

CertificatePolicy updatePolicy = new CertificatePolicy

{

X509CertificateProperties = new X509CertificateProperties

{

SubjectAlternativeNames = new SubjectAlternativeNames

{

DnsNames = new[] { "az204.examref.testing" }

}

}

};

Task.Run(async () => await keyVault.UpdateCertificatePolicyAsync(

vaultBaseURL, certName, updatePolicy)).Wait();

Task.Run(async () => await keyVault.CreateCertificateAsync(vaultBaseURL,

certName)).Wait();

Thread.Sleep(10000);

certificate = Task.Run(async () => await keyVault.GetCertificateAsync(

vaultBaseURL, certName)).

GetAwaiter().GetResult();

Task.Run(async () => await keyVault.UpdateCertificateAsync(certificate.

CertificateIdentifier.Identifier, null,

new CertificateAttributes { Enabled =

false })).Wait();

Thread.Sleep(10000);

// Delete the self-signed certificate.

Task.Run(async () => await keyVault.DeleteCertificateAsync(vaultBaseURL,

certName)).Wait();

// Manage keys in the Key Vault

string keyName = "key-az204";

NewKeyParameters keyParameters = new NewKeyParameters

{

Kty = "EC",

CurveName = "SECP256K1",

KeyOps = new[] { "sign", "verify" }

};

Task.Run(async () => await keyVault.CreateKeyAsync(vaultBaseURL, keyName,

keyParameters)).Wait();

var key = Task.Run(async () => await keyVault.GetKeyAsync(vaultBaseURL,

keyName)).GetAwaiter().GetResult();

// Update keys in the Key Vault

Task.Run(async () => await keyVault.UpdateKeyAsync(vaultBaseURL, keyName,

null, new KeyAttributes { Expires = DateTime.UtcNow.

AddYears(1)})).Wait();

key = Task.Run(async () => await keyVault.GetKeyAsync(vaultBaseURL,

keyName)).GetAwaiter().GetResult();

// Delete keys from the Key Vault

Task.Run(async () => await keyVault.DeleteKeyAsync(vaultBaseURL, keyName)).

Wait();

return View();

}

At this point, you should be able to run the example. Because you didn’t make any modifications to any view, you should not be able to see any changes in your Azure Key Vault. To be able to see how this code creates, reads, modifies, and deletes the different item types in your Azure Key Vault, you should set some breakpoints:

1. Add a breakpoint to the following lines:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg188a)

string secretName = "secret-az204";

string certName = "cert-az204";

string keyName = "key-az204";

1. Open your Azure Key Vault in the Azure Portal, as shown in step 9 of the previous procedure.
2. On your Azure Key Vault blade, click Secrets in the Settings section in the navigation menu.
3. In Visual Studio, press F5 to debug your project.
4. When you hit the breakpoint, press F10 and go back to the Azure portal to see the results. You should use the Refresh button to see the changes in your Azure Key Vault.

***NOTE* FORBIDDEN ACCESS**

If you get a Forbidden Access Error while you are debugging your application in Visual Studio, ensure that you created an Access Policy for the user account that you have configured in your Visual Studio for connecting with your Azure subscription. You need to ensure that the Access Policy grants all the needed privileges to the different object types in the Azure Key Vault. Ensure also that the account that you are using for developing is correctly authenticated in Azure Active Directory. Check your development account in Tools > Options > Azure Service Authentication. If there is a Re-Authenticate link below your development account, click the link to authenticate again .

When you work with the KeyVault API, you need to create a KeyVaultClient object that is responsible for the communication with the Azure Key Vault services. As described in the example in the “Implement Managed Service Identity (MSI)/Service Principal authentication” section, you need to get an access token for authenticating your service or user principal to the Azure Key Vault. The following code snippet shows how to perform this authentication:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg189-1a)

var azureServiceTokenProvider = new AzureServiceTokenProvider();

var keyVault = new KeyVaultClient(new KeyVaultClient.AuthenticationCallback(

azureServiceTokenProvider.KeyVaultTokenCallback));

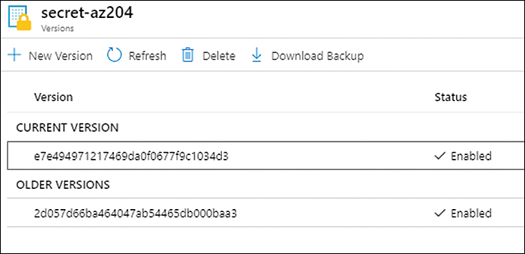
Now you can use the keyVault variable for working with the different item types. The KeyVault API provides specialized methods for each item type. This way, you should use the SetSecretAsync() method for creating a new secret in your Azure Key Vault. The following code snippet shows how to create a new secret:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg189-2a)

Task.Run(async () => await keyVault.SetSecretAsync(vaultBaseURL, secretName, "This is a

secret testing value")).Wait();

If you try to create a new secret, key, or certificate using the same name of an object that already exists in the vault, you are creating a new version of that object, as shown in [Figure 3-10](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig10). The only exception to this rule is if you have enabled soft-deletion in your Azure Key Vault, and you try to create a new secret using the same name as a deleted object. In that situation, you get a collision exception. You can click on each version to review the properties of the object for that version.



**Figure 3-10** A secret object with different versions

Most of the methods in the KeyVault API that work with items require the vault URL and the name of the item that you want to access. In this example, you define a variable with the correct value at the beginning of the Index() method, as shown in the following code snippet:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg189-3a)

string keyVaultName = "<YOUR\_VAULT's\_NAME>";

string vaultBaseURL = $"https://{keyVaultName}.vault.azure.net";

These methods are usually overloaded for accepting an object identifier instead of the vault base URL and the object’s name. The identifier has the following form:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg190-0a)

https://{*keyvault-name*}.vault.azure.net/{*object-type*}/{*object-name*}/{*object-version*}

Where:

* *Keyvault-name* is the name of the key vault where the object is stored.
* *Object-type* is the type of object that you want to work with. This value can be secrets, keys, or certificates.
* *Object-name* is the name that you give the object in the vault.
* *Object-version* is the version of the object that you want to access.

Creating a key or certificate uses a slightly different approach from the one that you used for creating a secret. Keys and certificates are more complex objects and require some additional configuration for creating them. The following code snippet extracted from [Listing 3-25](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex25) shows how to create a new self-signed certificate in the Azure Key Vault:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg190-1a)

// Create a new self-signed certificate

var policy = new CertificatePolicy

{

IssuerParameters = new IssuerParameters

{

Name = "Self",

},

KeyProperties = new KeyProperties

{

Exportable = true,

KeySize = 2048,

KeyType = "RSA"

},

SecretProperties = new SecretProperties

{

ContentType = "application/x-pkcs12"

},

X509CertificateProperties = new X509CertificateProperties

{

Subject = "CN=AZ204KEYVAULTDEMO"

}

};

Task.Run(async () => await keyVault.CreateCertificateAsync(vaultBaseURL, certName,

policy, new CertificateAttributes { Enabled = true })).Wait();

You need to create a CertificatePolicy object before you can create the certificate. A certificate policy is an object that defines the properties of how to create a certificate and any new version associated with the certificate object. You use this certificate policy object as a parameter of the CreateCertificateAsync() method. If you need to modify any property of an existing certificate, you need to define a new certificate policy, update the policy using the UpdateCertificatePolicyAsync() method, and create a new certificate version using the CreateCertificateAsync() method, as shown in the following code snippet:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg190-2a)

// Update properties associated with the certificate.

CertificatePolicy updatePolicy = new CertificatePolicy

{

X509CertificateProperties = new X509CertificateProperties

{

SubjectAlternativeNames = new SubjectAlternativeNames

{

DnsNames = new[] { "az204.examref.testing" }

}

}

};

Task.Run(async () => await keyVault.UpdateCertificatePolicyAsync(vaultBaseURL, certName,

updatePolicy)).Wait();

Task.Run(async () => await keyVault.CreateCertificateAsync(vaultBaseURL, certName))

.Wait();

Deleting an object from the key vault is quite straightforward; you only need to provide the vault base URL and the object’s name to the DeleteSecretAsync(), DeleteCertificateAsync(), or DeleteKeyAsync() method. Azure Key Vault also supports soft-delete operations on the protected objects or the vault itself. This option is enabled by default. When you soft delete an object or a vault, the Azure Key Vault provider automatically marks them as deleted but holds the object or vault for a default period of 90 days. This means you can recover the deleted object later if needed.

***NEED MORE REVIEW?* MORE DETAILS ABOUT KEYS, SECRETS, AND CERTIFICATES**

You can find more information about the details of the different object types that are available in the Azure Key Vault service by reviewing the article at [*https://docs.microsoft.com/en-us/azure/key-vault/about-keys-secrets-and-certificates*](https://docs.microsoft.com/en-us/azure/key-vault/about-keys-secrets-and-certificates).

**Images *EXAM TIP***

The kind of information that you usually store in an Azure Key Vault is essential information that needs to keep secret, like passwords, connection strings, private keys, and things like that. When configuring the access to your Key Vault, carefully review the access level you grant to the security principal. As a best practice, you should always apply the principle of least privilege. You grant access to the different levels in a Key Vault by creating Access Policies.

**Implement Managed Identities for Azure resources**

When you are designing your application, you usually identify the different services or systems on which your application depends. For example, your application may need to connect to an Azure SQL database for storing data or may need to connect to Azure Event Hub for reading messages from other services. In all these situations, there is a common need to authenticate with the service before you can access it. In the Azure SQL database case, you need to use a connection string; if you need to connect to an Azure Event Hub, you need to use a combination of event publishers and Shared Access Signature (SAS) tokens.

The drawback of this approach is that you need to store a security credential, token, or password to be able to authenticate to the service that you want to access. This is a drawback because you might find that this information is stored on developers’ computers or is checked in to the source control by mistake. You can address most of these situations by using the Azure Key Vault, but your code still needs to authenticate to Azure Key Vault to get the information for accessing the other services.

Fortunately, Azure Active Directory (Azure AD) provides the Managed Identities for Azure resources (formerly known as Managed Service Identity) that removes the need to use credentials for authenticating your application to any Azure service that supports Azure AD authentication. This feature automatically creates a managed identity that you can use for authenticating to any service that supports Azure AD authentication without needing to provide any credential.

When you work with managed identities, you can work with two different types:

* **System-assigned managed identities** These are identities that Azure automatically enables when you create an Azure service instance, like an Azure virtual machine (VM) or an Azure data lake store. Azure creates an identity associated with the new instance and stores it to the Azure AD tenant associated with the subscription where you created the service instance. If you decide to delete the service instance, then Azure automatically deletes the managed instance associated with the service instance stored in the Azure AD tenant.
* **User-assigned managed identities** You can create your managed identities in the Azure AD tenant associated with your Azure subscription. You can associate this type of managed identity to one or more service instances. The lifecycle of the managed identity is independent of the service instance. This means that if you delete the service instance, the user-assigned managed identity remains in the Azure AD tenant. You need to remove the managed identity manually.

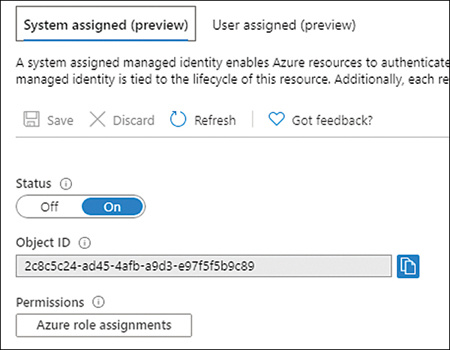
Usually, you use the system-assigned managed identities when your workload is contained within the same Azure resource, or you need to independently identify each of the service instances, like Virtual Machines. On the other hand, if you need to grant access to a workload that is distributed across different resources or you need to pre-authorize a resource as part of a provisioning flow, you should use user-assigned managed identities.

When you work with managed identities, you need to bear in mind three concepts:

* **Client ID** This is a unique identifier generated by Azure AD. This ID associates the application and the service principal during its initial provisioning.
* **Principal ID** This is the ID of the service principal associated with the managed identity. A service principal and a managed identity are tightly coupled, but they are different objects. The service principal is the object that you use to grant role-based access to an Azure resource.
* **Azure Instance Metadata Service (IMDS)** When you use managed identities in an Azure VM, you can use the IMDS for requesting an OAuth Access Token from your application deployed within the VM. The IMDS is a REST endpoint that you can access from your VM using a nonroutable IP address (169.254.169.254).

The following example shows how to create a system-assigned identity in an Azure App Service and how to use this managed identity from your code for accessing an Azure Key Vault. For this example, you need to have an empty Azure App Service, an Azure Key Vault, and at least one item on the Azure Key Vault. You also need to have your Visual Studio connected to the Azure subscription where you have configured the Azure Key Vault.

1. Open the Azure Portal at [*https://portal.azure.com*](https://portal.azure.com/).
2. In the search text box at the top of the Azure portal, type the name of your Azure Web App. If you don’t have an Azure Web App, you can create a new Azure Web App by using the procedure at [*https://docs.microsoft.com/en-in/azure/app-service/app-service-web-get-started-dotnet*](https://docs.microsoft.com/en-in/azure/app-service/app-service-web-get-started-dotnet).
3. On the Azure Web App Service blade, click the Identity menu item in the Settings section.
4. On the Status switch control, click the On option.
5. Click the Save button.
6. In the Enable System Assigned Managed Identity dialog box, click the Yes button.
7. Once you enable the system-assigned managed identity, you get the Principal or Object ID, as shown in [Figure 3-11](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03fig11).



**Figure 3-11** System assigned managed identity

1. Open Visual Studio 2019.
2. In the welcome window of Visual Studio 2019, on the Get Started column, click Create A New Project.
3. On the Create A New Project window, on the drop-down menu, All Languages drop-down menu, select C#.
4. In the Search For Templates text box type **asp.net**.
5. In the result list, click ASP.NET Web Application (.NET Framework).
6. Click the Next button at the bottom right of the window.
7. In the Configure Your New Project, type a Project Name, a Location, and a Solution Name for your project.
8. Click the Create button at the bottom right of the window.
9. In the Create A New ASP.NET Web Application window, select the MVC template on the template list in the middle of the left side of the window. MVC is for Model-View-Controller.
10. On the right side of the Create A New ASP.NET Web Application window, on the Authentication section, ensure the Authentication is set to No Authentication.
11. Click the Create button at the bottom right of the window.
12. In the Visual Studio window, click Tools > NuGet Package Manager > Manage NuGet Packages For Solution.
13. On the NuGet Package Manager tab, click Browse.
14. Type **Microsoft.Azure.Services.AppAuthentication** and press Enter.
15. Click the Microsoft.Azure.Services.AppAuthentication package.
16. On the right side of the NuGet Manager tab, click the check box next to your project.
17. Click the Install button.
18. In the Preview Changes window, click OK.
19. In the License Acceptance window, click the I Accept button.
20. Repeat steps 20 through 26 and install the Microsoft.Azure.KeyVault package.
21. Open the HomeController.cs file in the Controllers folder.
22. Add the following statements to the HomeController.cs file:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#pg194a)

using Microsoft.Azure.KeyVault;

using Microsoft.Azure.Services.AppAuthentication;

using System.Threading.Tasks;

1. Replace the content of the Index() method with the content of [Listing 3-26](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex26). The crucial pieces of code related to accessing the Azure Key Vault are highlighted in bold.

**Listing 3-26** Getting a secret from the key vault

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-26a)

// C#. ASP.NET.

string keyVaultName = "<PUT\_YOUR\_KEY\_VAULT\_NAME\_HERE>";

string secretName = "<PUT\_YOUR\_SECRET\_NAME\_HERE>";

//Get a token for accessing the Key Vault.

**var azureServiceTokenProvider = new AzureServiceTokenProvider();**

//Create a Key Vault client for accessing the items in the vault.

var keyVault = new KeyVaultClient(new KeyVaultClient.AuthenticationCallback(

**azureServiceTokenProvider.KeyVaultTokenCallback**));

var secret = Task.Run(async () => await keyVault.GetSecretAsync(

$"https://{keyVaultName}.vault.azure.net/secrets/{secretName}"))

.GetAwaiter().GetResult();

ViewBag.KeyVaultName = keyVaultName;

ViewBag.keyName = secretName;

ViewBag.secret = secret.Value;

return View();

As you can see, this code is quite similar to the code in [Listing 3-25](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex25). The reason is that you used managed identities to get access to the Azure Key Vault in the example in [Listing 3-25](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex25). Before you can access the Azure Key Vault, you need to get an OAuth token by using the AzureServiceTokenProvider class. Then you can create your Azure Key Vault client and get any item stored in the vault. When you create the Azure Key Vault client, make sure you provide the KeyVaultTokenCallback. Even if you get a valid access token, you still need to grant access to your Azure App Service application in the Azure Key Vault.

1. Open the Views > Home > Index.cshtml file.
2. Append the content of [Listing 3-27](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03ex27) to the end of the file.

**Listing 3-27** Adding secret information to the home page

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03_images.xhtml#lis3-27a)

// C#. ASP.NET.

<div class="row">

<div class="col-lg-12">

<dl class="dl-horizontal">

<dt>Key Vault Name: </dt>

<dd>@ViewBag.keyVaultName</dd>

<dt>Key Name: </dt>

<dd>@ViewBag.keyName</dd>

<dt>Key Secret: </dt>

<dd>@ViewBag.secret</dd>

</dl>

</div>

</div>

At this point, you could run your project and see the results. Depending on the access policies defined in your Azure Key Vault, your Azure user may already have access to the secrets stored in the key vault. In that case, you should be able to access the secret stored in the Azure Key Vault. If you get an exception when running the web application, there are good chances that you don’t have access to the Azure Key Vault. The following steps show how to grant access to your Azure App Service application in the Azure Key Vault.

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. Type the name of your Azure Key Vault in the search text box at the top of the Azure portal. If you don’t already have an Azure Key Vault and need to create a new one, you can use the procedure at [*https://docs.microsoft.com/en-us/azure/key-vault/quick-create-portal*](https://docs.microsoft.com/en-us/azure/key-vault/quick-create-portal).
3. On your Azure Key Vault blade, click Access Policies in the Settings section.
4. On the Access Policies blade, click Add New.
5. On the Add Access Policy page, select Secret Management in the Configure From Template drop-down menu.
6. Click the Select Principal control.
7. In the Principal panel, type the name of your Azure App Service in the Select text box. Your Azure App Service should appear on the list below the text box.
8. Click your App Service name in the list below the Select text box.
9. Click the Select button at the bottom of the panel.
10. Click the Add button at the bottom of the Add Access Policy blade.
11. Click the Save button at the top of the Access Policies blade.
12. In the Visual Studio window, right-click your project’s name in the Solution Explorer window.
13. In the contextual menu, click Publish.
14. In the Pick A Publish Target window, ensure that App Service is selected on the left side of the window.
15. In the Azure App Service section, click Select Existing.
16. Click the Create Profile button at the bottom-right corner of the window.
17. In the App Service window, in the tree view at the bottom of the window, look for your App Service and click it.
18. Click the OK button.

At this point, Visual Studio starts publishing your web application to the selected Azure App Service. When the publishing operation finishes, you should be able to see your web application showing the content of the secret stored in your Key Vault.

**Images *EXAM TIP***

You can configure two different types of managed identities: system- and user-assigned. System-assigned managed identities are tied to the service instance. If you delete the service instance, the system-assigned managed identity is automatically deleted as well. You can assign the same user-assigned managed identities to several service instances.

**Chapter summary**

* Authentication is the act of proving that a user is who he or she claims to be.
* A user authenticates by providing some information that the user only knows.
* There are several mechanisms of authentication that provide different levels of security.
* Some of the authentication mechanisms are form-based, token-based, or certificate-based.
* Using form-based authentication requires your application to store your users’ passwords.
* Form-based authentication requires HTTPS to make the authentication process more secure.
* Using token-based authentication, you can delegate the authorization to third-party authentication providers.
* You can add social logins to your application by using token-based authentication.
* Multifactor authentication is an authentication mechanism that requires the users to provide more than one piece of information that only the user knows.
* You can easily implement multifactor authentication by using Azure Active Directory.
* There are four main actors in OAuth authentication: client, resource server, resource owner, and authentication server.
* The resource owner needs to authenticate the client before sending the authorization grant.
* The access token grants access to the resource hosted on the resource server.
* The authorization grant or authorization code grants the client the needed rights to request an access token to the authorization server.
* The client uses the refresh token to get a new access token when it expires without needing to request a new authorization code.
* The JSON web token is the most extended implementation of OAuth tokens.
* Shared Access Signatures (SAS) is an authentication mechanism for granting access to Azure Storage Accounts without sharing account keys.
* Shared Access Signatures (SAS) tokens must be signed.
* There are three types of SAS token: user delegation, account, and service SAS.
* User delegation SAS tokens are signed using a key assigned to an Azure Active Directory user.
* Account and Service SAS are signed using the Azure Storage account key.
* You can hide the details of the SAS tokens from the URL by using Stored Access Policies.
* Shared access signature tokens provide fine-grained access control to your Azure storage accounts.
* You can create an SAS token for service, container, and item levels.
* You need to register applications in Azure Active Directory for being able to authenticate users using your tenant.
* There are three account types supported for authentication: accounts only in the organizational directory, accounts in any organizational directory, and Microsoft accounts.
* You need to provide a return URL for authenticating your application when requesting user authentication.
* You need to configure a secret or a certificate when your application needs to access information in other APIs.
* Role-Based Access Control (RBAC) authorization provides fine-grained access control to the resources.
* A security principal is an entity to which you can grant privileges.
* Security principals are users, groups, service principals, and managed identities.
* A permission is an action that a security principal can make with a resource.
* A role definition, or role, is a group of permissions.
* A scope is a level where you can assign a role.
* A role association is a relationship between a security principal, a role, and a scope.
* There are four scopes: management groups, subscription, resource group, and resources.
* You can centralize the configuration of your distributed application using Azure App Configuration.
* Azure App Configuration stores the information using key-value pairs.
* Values in the Azure App Configuration are encrypted.
* Azure Key Vault provides better security than the Azure App Configuration service.
* The limit of size for an Azure App Configuration is 10,000, including the key, label, and value.
* You can create references from Azure App Configuration items to Azure Key Vault items.
* Azure Key Vault allows you to store three types of objects: keys, secrets, and certificates.
* You should use managed identities authentication for accessing the Azure Key Vault.
* You need to define a certificate policy before creating a certificate in the Azure Key Vault.
* If you import a certificate into the Azure Key Vault, a default certificate policy is automatically created for you.

**Thought experiment**

In this thought experiment, demonstrate your skills and knowledge of the topics covered in this chapter. You can find answers to this thought experiment in the next section.

You are developing a web application for your company. The application is in the early stages of development. This application is an internal application that will be used only by the employees 199of the company. Your company uses Office 365 connected with your company’s Active Directory domain. The application needs to use information from Office 365. Answer the following questions about the security implementation of this application:

[**1.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03que1a) The employees need to be able to access the application using the same username and password they use for accessing Office 365. What should you do?

[**2.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03que2a) You are using Azure App Services for developing the application. You need to ensure that the web application can access other Azure services without using credentials in your code. What should you do?

[**3.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03que3a) You need to ensure that the configuration of your application is stored in central storage. You also need to provide the best security for sensitive information like connection strings and passwords. What should you do?

**Thought experiment answers**

This section contains the solution to the thought experiment. Each answer explains why the answer choice is correct.

[**1.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03que1) You should use OAuth authentication with Azure Active Directory (Azure AD). If you want your application to be able to use Azure AD OAuth authentication, you need to register your application in your Azure AD tenant. Because your application needs access to information in Office 365, you also need to create a client secret before you can access the Microsoft Graph API. When you connect Office 365 with an Active Directory (AD) domain, users in the AD domain can authenticate to Office 365 using the same username and password they use in the AD domain. Office 365 uses an Azure AD tenant for managing the identities of the users in the subscription. Your organization has already configured the synchronization between AD and Office 365 and Azure AD. By using OAuth authentication with Azure AD, your users should be able to access your application using the same username and passwords that they use in the AD domain.

[**2.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03que2) You should use the Managed Service Identity (MSI) authentication. Using the feature, Azure authenticates services based on a service principal configured in a service instance. You can use MSI authentication with services that support Azure AD authentication, like Azure Key Vault or Azure SQL Databases. You need to enable a system-assigned or user-assigned managed identity on your Azure App Service. Using MSI, the Azure SQL Database authenticates the identity assigned to your Azure App Service without needing you to provide any password.

[**3.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch03.xhtml#ch03que3) You should create an Azure App Configuration store. This is the appropriate service for securely storing your app configuration settings in centralized storage. Although the Azure App Configuration store provides secure storage by encrypting the value of the key-value pairs representing your settings, you should use Key Vault references in your Azure App Configuration store for that sensitive information that requires a higher level of security. Azure Key Vault uses hardware-based encryption for storing keys, secrets, and certificates.