

Add authentication to your Windows app

7 minutes to
read

Contributors      [all](#)

This topic shows you how to add cloud-based authentication to your mobile app. In this tutorial, you add authentication to the Universal Windows Platform (UWP) quickstart project for Mobile Apps using an identity provider that is supported by Azure App Service. After being successfully authenticated and authorized by your Mobile App backend, the user ID value is displayed.

This tutorial is based on the Mobile Apps quickstart. You must first complete the tutorial [Get started with Mobile Apps](#).

Register your app for authentication and configure the App Service

First, you need to register your app at an identity provider's site, and then you will set the provider-generated credentials in the Mobile Apps back end.

1. Configure your preferred identity provider by following the provider-specific instructions:

- [Azure Active Directory](#)
- [Facebook](#)
- [Google](#)
- [Microsoft](#)
- [Twitter](#)

2. Repeat the previous steps for each provider you want to support in your app.

Add your app to the Allowed External Redirect URLs

Secure authentication requires that you define a new URL scheme for your app. This allows the authentication system to redirect back to your app once the authentication process is complete. In this tutorial, we use the URL scheme *appname* throughout. However, you can use any URL scheme you choose. It should be unique to your mobile application. To enable the redirection on the server side:

1. In the [Azure portal], select your App Service.
2. Click the **Authentication / Authorization** menu option.
3. In the **Allowed External Redirect URLs**, enter `url_scheme_of_your_app://easyauth.callback`. The **url_scheme_of_your_app** in this string is the URL Scheme for your mobile application. It should follow normal URL specification for a protocol (use letters and numbers only, and start with a letter). You should make a note of the string that you choose as you will need to adjust your mobile application code with the URL Scheme in several places.
4. Click **OK**.
5. Click **Save**.

Restrict permissions to authenticated users

By default, APIs in a Mobile Apps back end can be invoked anonymously. Next, you need to restrict access to only authenticated clients.

- **Node.js back end (via the Azure portal):**

In your Mobile Apps settings, click **Easy Tables** and select your table. Click **Change permissions**, select **Authenticated access only** for all permissions, and then click **Save**.

- .NET back end (C#):

In the server project, navigate to **Controllers** > **TodoItemController.cs**. Add the `[Authorize]` attribute to the **TodoItemController** class, as follows. To restrict access only to specific methods, you can also apply this attribute just to those methods instead of the class. Republish the server project.

```
[Authorize]
public class TodoItemController : TableController<TodoItem>
```

- Node.js backend (via Node.js code):

To require authentication for table access, add the following line to the Node.js server script:

```
table.access = 'authenticated';
```

For more details, see [How to: Require authentication for access to tables](#). To learn how to download the quickstart code project from your site, see [How to: Download the Node.js backend quickstart code project using Git](#)

Now, you can verify that anonymous access to your backend has been disabled. With the UWP app project set as the start-up project, deploy and run the app; verify that an unhandled exception with a status code of 401 (Unauthorized) is raised after the app starts. This happens because the app attempts to access your Mobile App Code as an unauthenticated user, but the *TodoItem* table now requires authentication.

Next, you will update the app to authenticate users before requesting resources from your App Service.

Add authentication to the app

1. In the UWP app project file `MainPage.xaml.cs` and add the following code snippet:

```
// Define a member variable for storing the signed-in user.
private MobileServiceUser user;

// Define a method that performs the authentication process
// using a Facebook sign-in.
private async System.Threading.Tasks.Task<bool> AuthenticateAsync()
{
    string message;
    bool success = false;
    try
    {
        // Change 'MobileService' to the name of your MobileServiceClient instance.
        // Sign-in using Facebook authentication.
        user = await App.MobileService
            .LoginAsync(MobileServiceAuthenticationProvider.Facebook, "{url_scheme_of_your_app}");
        message =
            string.Format("You are now signed in - {0}", user.UserId);

        success = true;
    }
    catch (InvalidOperationException)
    {
        message = "You must log in. Login Required";
    }

    var dialog = new MessageDialog(message);
    dialog.Commands.Add(new UICommand("OK"));
    await dialog.ShowAsync();
    return success;
}
```

This code authenticates the user with a Facebook login. If you are using an identity provider other than Facebook, change the value of **MobileServiceAuthenticationProvider** above to the value for your provider.

2. Replace the **OnNavigatedTo()** method in MainPage.xaml.cs. Next, you will add a **Sign in** button to the app that triggers authentication.

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    if (e.Parameter is Uri)
    {
        App.MobileService.ResumeWithURL(e.Parameter as Uri);
    }
}
```

3. Add the following code snippet to the MainPage.xaml.cs:

```
private async void ButtonLogin_Click(object sender, RoutedEventArgs e)
{
    // Login the user and then load data from the mobile app.
    if (await AuthenticateAsync())
    {
        // Switch the buttons and load items from the mobile app.
        ButtonLogin.Visibility = Visibility.Collapsed;
        ButtonSave.Visibility = Visibility.Visible;
        //await InitLocalStoreAsync(); //offline sync support.
        await RefreshTodoItems();
    }
}
```

4. Open the MainPage.xaml project file, locate the element that defines the **Save** button and replace it with the following code:

```

<Button Name="ButtonSave" Visibility="Collapsed" Margin="0,8,8,0"
    Click="ButtonSave_Click">
    <StackPanel Orientation="Horizontal">
        <SymbolIcon Symbol="Add"/>
        <TextBlock Margin="5">Save</TextBlock>
    </StackPanel>
</Button>
<Button Name="ButtonLogin" Visibility="Visible" Margin="0,8,8,0"
    Click="ButtonLogin_Click" TabIndex="0">
    <StackPanel Orientation="Horizontal">
        <SymbolIcon Symbol="Permissions"/>
        <TextBlock Margin="5">Sign in</TextBlock>
    </StackPanel>
</Button>

```

5. Add the following code snippet to the App.xaml.cs:

```

protected override void OnActivated(IActivatedEventArgs args)
{
    if (args.Kind == ActivationKind.Protocol)
    {
        ProtocolActivatedEventArgs protocolArgs = args as ProtocolActivatedEventArgs;
        Frame content = Window.Current.Content as Frame;
        if (content.Content.GetType() == typeof(MainPage))
        {
            content.Navigate(typeof(MainPage), protocolArgs.Uri);
        }
    }
    Window.Current.Activate();
    base.OnActivated(args);
}

```

6. Open Package.appxmanifest file, navigate to **Declarations**, in **Available Declarations** dropdown list, select **Protocol** and click **Add** button. Now configure the **Properties** of the **Protocol** declaration. In **Display name**, add the name you wish to display to users of your application. In **Name**, add your {url_scheme_of_your_app}.
7. Press the F5 key to run the app, click the **Sign in** button, and sign into the app with your chosen identity provider. After your sign-in is successful, the app runs without errors and you are able to query your backend and make updates to data.

Store the authentication token on the client

The previous example showed a standard sign-in, which requires the client to contact both the identity provider and the App Service every time that the app starts. Not only is this method inefficient, you can run into usage-relates issues should many customers try to start you app at the same time. A better approach is to cache the authorization token returned by your App Service and try to use this first before using a provider-based sign-in.

Note

You can cache the token issued by App Services regardless of whether you are using client-managed or service-managed authentication. This tutorial uses service-managed authentication.

1. In the MainPage.xaml.cs project file, add the following **using** statements:

```

using System.Linq;
using Windows.Security.Credentials;

```

2. Replace the **AuthenticateAsync** method with the following code:

```

private async System.Threading.Tasks.Task<bool> AuthenticateAsync()
{
    string message;
    bool success = false;

    // This sample uses the Facebook provider.
    var provider = MobileServiceAuthenticationProvider.Facebook;

    // Use the PasswordVault to securely store and access credentials.
    PasswordVault vault = new PasswordVault();
    PasswordCredential credential = null;

    try
    {
        // Try to get an existing credential from the vault.
        credential = vault.FindAllByResource(provider.ToString()).FirstOrDefault();
    }
    catch (Exception)
    {
        // When there is no matching resource an error occurs, which we ignore.
    }

    if (credential != null)
    {
        // Create a user from the stored credentials.
        user = new MobileServiceUser(credential.UserName);
        credential.RetrievePassword();
        user.MobileServiceAuthenticationToken = credential.Password;

        // Set the user from the stored credentials.
        App.MobileService.CurrentUser = user;

        // Consider adding a check to determine if the token is
        // expired, as shown in this post: http://aka.ms/jww5vp.

        success = true;
        message = string.Format("Cached credentials for user - {0}", user.UserId);
    }
    else
    {
        try
        {
            // Sign in with the identity provider.
            user = await App.MobileService
                .LoginAsync(provider, "{url_scheme_of_your_app}");

            // Create and store the user credentials.
            credential = new PasswordCredential(provider.ToString(),
                user.UserId, user.MobileServiceAuthenticationToken);
            vault.Add(credential);

            success = true;
            message = string.Format("You are now signed in - {0}", user.UserId);
        }
        catch (MobileServiceInvalidOperationException)
        {
            message = "You must sign in. Sign-In Required";
        }
    }

    var dialog = new MessageDialog(message);
    dialog.Commands.Add(new UICommand("OK"));
    await dialog.ShowAsync();

    return success;
}

```

In this version of **AuthenticateAsync**, the app tries to use credentials stored in the **PasswordVault** to access the service. A regular sign-in is also performed when there is no stored credential.

Note

A cached token may be expired, and token expiration can also occur after authentication when the app is in use. To learn how to determine if a token is expired, see [Check for expired authentication tokens](#). For a solution to handling authorization errors related to expiring tokens, see the post [Caching and handling expired tokens in Azure Mobile Services managed SDK](#)

3. Restart the app twice.

Notice that on the first start-up, sign-in with the provider is again required. However, on the second restart the cached credentials are used and sign-in is bypassed.


Next steps



Now that you completed this basic authentication tutorial, consider continuing on to one of the following tutorials:

- [Add push notifications to your app](#)
Learn how to add push notifications support to your app and configure your Mobile App backend to use Azure Notification Hubs to send push notifications.
- [Enable offline sync for your app](#)
Learn how to add offline support your app using an Mobile App backend. Offline sync allows end-users to interact with a mobile app—viewing, adding, or modifying data—even when there is no network connection.


Feedback

We'd love to hear your thoughts. Choose the type you'd like to provide:

Product feedback 

 Sign in to give documentation feedback 

Our new feedback system is built on GitHub Issues. Read about this change in [our blog post](#).

 Loading feedback...



[Previous Version Docs](#) • [Blog](#) • [Contribute](#) • [Privacy & Cookies](#) • [Terms of Use](#) • [Site Feedback](#) • [Trademarks](#)