Xamarin studio tutorial pdf

Are you a .NET developer who has always wanted to make a mobile app? Or maybe you tried building native mobile apps with Android or iOS, but didn't like the languages? Well, then, you're in luck! The .NET world was blessed with Xamarin; a set of tools that lets you build mobile apps for Android, iOS, and Windows within Visual Studio. Xamarin has two main flavors: Xamarin platform (Xamarin.iOS and Xamarin.Android) and Xamarin.Forms. With Xamarin.Forms, the vast majority of your business logic and user interface can be written within a shared project that produces fully functioning applications on all 3 iOS, Android, and Windows (UWP) operating systems. The Xamarin platform, on the other hand, is a very platform-specific job and is more like writing native applications, but with C#. In this tutorial, I'll be looking more closely at the Xamarin platform and the Android operating system known as Xamarin.Android. The overall goal is to allow you to create a simple native Android app with basic user authentication included. Set up Visual Studio and your environment To keep track, you'll need a copy of Visual Studio in addition to the 'Mobile with' development. NET workload. You can enable this feature from the first visual studio installation or access it from the Tools -> get tools and features...' Menu item: When testing and running your app, you have the option to do so with an Android emulator running on your development machine or connecting directly to an existing Android device. There is no right option here and different developers prefer different form factors. If you choose the previous option, you need to ensure that once you have selected the workload that in the right pane ('Installation Details') the check boxes for Intel Hardware Accelerated Execution Manager and Google Android Emulator are selected (as seen above). Check your Android environment in Visual Studio To make sure everything has been installed correctly and has been set up correctly, go to 'Tools -> Xamarin -> Android Settings' and verify that the Java Development Kit and Android SDK Location paths are valid (i.e. have a green tick): If it's missing, you'll have to manually install the Java Development Kit or Android SDK, respectively. Create a Xamarin App Start by creating a new project and select the master model 'Android App (Xamarin)) (found in the Android menu). On the next page you will want to choose the 'Single View App' option as it is a great initial template to work with. Regarding the Minimum Version of Android, this is something that comes down to your personal choice as a developer. There is an exchange here between being able to access the latest and greatest API features in newer versions and customers who have older versions. To help you make this decision, Google publishes platform version distribution data that it collects as part of its distribution dashboard at a fairly regular rate. My personal personal preference between 5.0 or 6.0, depending on whether this is an app for public consumption or a commissioned application only for corporate phones (i.e. they will probably have the latest updates); in this example I went with the latter. Note that this version differs from the target version of Android and should always be set to the latest SDK version released, as nothing less will be accepted in the Google Play store. Once you have created this all that remains is to import the necessary NuGet packages. For this tutorial you will need: Xamarin.OpenId.AppAuth.Android - To authenticate the user you will be using the OpenID Connect standard (an enhancement of OAuth 2.0). The easiest way to implement the client code that has computed this specification is by using the AppAuth client SDK for Android, and with help, Xamarin has a package of this functionality available for you to use. System.IdentityModel.Tokens.Jwt - The authentication method here uses JSON Web Tokens. To extract the required data from these tokens, you will need this package. In addition, you will need these two packages as well as they are trusted by AppAuth: Xamarin.Android.Support.v4 Xamarin.Android.Support.Support.CustomTabs Familiarize yourself with the Xamarin Project If you have never worked with Android before, one of the fundamental principles to put your head around is the concept of an Activity. Activities are components used to display your user interface; in its most basic form, you can think of Activities as being equal to pages in an application that the user can navigate between. An activity in code is represented by a class, however, as a page in ASP.NET you can (and almost always will) associate an XML-based layout file (a .axml file) with it for a displayable design. All new projects create a 'MainActivity.cs' and 'activity_main.axml' file to begin as the first activity (i.e. page) to run when opening the application. This can be changed to any other Activity by using the MainLauncher = true property within the Class Activity attribute. The features are designed to be handled within your own directory and follow a somewhat strict naming convention. I strongly recommend storing both of your resources and is feasible in this directory, since it simplifies the reuse of these variables for their continuous development. In the 'values' directory of the Resources directory is where you will find files for specific purposes: Strings.xml - Hosts all strings facing the user. This is especially important to use as it allows you to locate your strings to a global audience. Styles.xml - Where you'll find the attributes to style your design objects; think of it as a CSS file. Colors.xml - A place to store references to colors you use most often as part of your style. Dimens.xml - How name may imply, where you define dimensions defined for the layout of your application. The other notable directories do not follow any naming convention for your files, but they must contain certain file types: Layout - - to store your .axml files. These files define complete activity layouts, layout components that you programmatically generate and populate, dialog layouts (alert) etc. Menu - Where you'll find menu definitions and their items. These are .xml files that have menu as root element and children's elements of items, of which they can be grouped with group elements. The most common menus you'll find are the overflow menu (from the three vertical dots button) or the navigation menu (from the domestic 'hamburger' button). Mipmap - Where you want to define images that need to be scaled depending on the density of the screen, that is, those referenced by other applications, and not used internally. The application icon is the most common content you would put in mipmap directories. Drawable - It's not standard in a standard project template, but it can be created yourself. This is where you store your drawable images/content to be used within the app, for example, splash screen, custom progress bar designs, icons used within the app, etc. Finally, if you have some raw content file that you want to use as part of your application (for example, a text file or font), then the Assets directory is where to place them. As Assets, these files will be deployed with your app for you to access with Asset Manager. To learn more about Assets and Resources and how to use them, there are useful 'About' text files in each directory of a newly created project. Add user authentication to Xamarin with OpenID Connect Most applications these days require some form of user IDENTIFICATION to enable the developer to offer tailored experiences and, in turn, allow the user to keep their data across devices/installations. In addition, there is the issue of access control that can be useful for authorizing a subset of users for extra functionality. Of course, this can be a very laborious task, especially if you are in the business of building applications and you will need a new system for each. Luckily with okta you can set up an app in mere minutes and then all the hard work is done for you! The Okta service is compatible with OAuth 2.0 and a certified OpenID Connection Provider, and thus works seamlessly with the AppAuth client SDK for all your authentication and authorization needs. Set up your Okta app First, you must set up a new app in your Okta account for this project. If you don't already have one, it's very easy to create a new developer account forever free. Once this is complete and you have logged into the developer dashboard, make a note of the Org URL, as we'll need it later: From the control panel go to the 'Apps' tab and from there 'Add app'. You creating a native app for Android, so it's best to choose the 'Native' platform template. From this page add a name to your app and leave everything as standard. Once saved, take note of your login redirect URIs and customer ID, as you'll be needing these next ones. So you can use these three Okta values with ease in future, I would recommend putting them in their own static class within a new directory for authentication logic: public static class configuration { public const chain ClientId = {yourClientId}; public const track LoginRedirectUri = {yourRedirectUri}; public banner const OrgUrl = https://{yourOktaDomain}; } Create the Authentication Provider Let's get the boiler code out of the way. You need to configure the application to inform you of your redirected URI schema. The scheme is your login redirect URI (in the default reverse domain name form) without the path. For example, from the screenshot above my scheme would be 'com.oktapreview.dev-123456'. The easiest way to do this is to insert the snippet of the intent filter below into your AndroidManifest.xml file in the Properties folder of your solution. Add the following XML in the App tag and <activity android:name=net.openid.appauth.RedirectUriReceiverActivity> <intent-filter> <action android:name=android.intent.action.VIEW></action> <category android:name=android.intent.category.DEFAULT></category> <category android:name=android.intent.category.BROWSABLE></category> <data android:scheme={yourRedirectRriScheme}></data> </intent-filter> </activity> change the value of the scheme for yours: You also need to set the template for the result of your authorization with a simple class. Although I'm not using all the values I wrote below within the code, I'll show you how to fill them out for you to use later. Because this is part of your application's model, create a folder called Models and add a AuthorizationResult.cs within it. Then add the following code: Public class authorizationResult { public chain AccessToken { get; set; } public string IdentityToken { get; set; } public bool IsAuthorized { get; set; } public string RefreshToken { get; set; } public string Scope { get; set; } } Android doesn't have a global state for you to work with, so if you want to pass simple values between activities, the best way to do that is with the Extras functionality in the Intent object. An Intent is a default class on Android and another central concept to understand. It is the abstraction of an operation to be performed (that is, its 'intentions'), and to navigate forward to another activity you need to create an instance of this class with which activity you 'intend' to go. The Extra properties in the Intent object are actually just a key dictionary for object values and are accessed by the "Get' and typed methods. While these methods keep use relatively clear and easy, I personally like to keep all access to them within their own class (to be precise, a class of extensions), to maintain a better separation of concerns. This is extremely useful because you don't need to access the keys classes and can ensure the safety of the type by placing and obtaining these values. At your authorization provider you'll be wanting: store AuthState, can check if it's there and return it if it is. Create a new one called extensions at the root of the solution. Then add a new class called IntentExtensions.cs. Make the class public and static, then add the following code within the class: public const range AuthStateKey = authState; GetAuthStateExtra public static chain (this intent of intent) { return intent. GetStringExtra(AuthStateKey); } public static bool HasAuthStateExtra (this intent of intent) { return intent. HasExtra (AuthStateKey); } static empty public PutAuthStateExtra (this intent intent, AuthState authState) { intent. PutExtra(AuthStateKey, authState.JsonSerializeString()); } Public static bool TryGetAuthStateFromExtra (this intent intent, out AuthState authState) { authState = null; if (!intent. HasAuthStateExtra()) { false return; } try { authState = AuthState.JsonDeserialize (intent. GetAuthStateExtra()); } catch (JSONException) { false return; } return authState != null; } Now it's time to set the authorization provider, AuthorizationProvider.cs in the Authentication folder that you created earlier for the Configuration.cs class. First, remove all usage statements within the newly created class, and then declare the configuration as static variables only: only uri private static read configurationendpoint = Uri.Parse(${Configuration.OrgUrl}/oauth2/default/.well-known/openid-configuration); private static read string[] Scopes = new[] { openid, profile, email, offline_access }; The configuration endpoint is a default in OpenID as the endpoint of discovery to find everything that is supported. Note here that I wrote this is using the provider name 'default'. If you have a different provider, you'll want to change that here. Also note that this is using the Android.Net flavor of the Uri class, not the System version - you'll need to add the first to its uses or fully qualify the type for this to work. The Scopes variable, like any other OpenID system, defines what we are allowed to access. Next, you must declare your member variables: authorization of private authorizationRequest authorization; Pending Private Intent CompletedIntent; authStateState private authorization; authorization authorization of private reading authorizationSS) Authorization service; context of private reading context; private reading tasks only<bool> TaskCompletionSourceCompletionSourceSource = new TaskCompletionSource<bool>(); A quick explanation about each: The authorization request and the completed intent are parameters created for use in performing the authorization call. I wrote them as global variables here to minimize the amount of pass parameters in different methods. The authorization variable State as it is named defines the current authorization state. The authorization variableServ contains an instance of the authorization service. The context variable is from the calling activity, so you can reference it when needed. Finally, the CompletionSource task allows you to make all of these calls asynchronously and then return once completed. Now you should set the values of these readonly</bool> </bool> </bool> in its constructor, and declare the public methods that your code will call: Public AuthorizationProvider(context) { AuthorizationService = new AuthorizationService(context); this.context = context; } public async Task<AuthorizationResult> SignInAsync() { ... } empty public NotifyCallback (Intent of Intent) { } The SignInAsync method is how you must have guessed an asynchronous method for signing a user. This returns the AuthorizationResult class that you wrote earlier. NotifyCallback, on the other hand, is for the call activity, once it has returned from the external signal on the page, to call back to the authorization provider and let you know it is done. The login method that I divided into several subroutines, and looks like this: Authorization Service Confirmation ServiceConfiguration = wait for AuthorizationService Configuration.FetchFromUrlAsync(ConfigurationEndpoint); BuildAuorizationRequest (service configuration); BuildCompletedIntent (serviceConfiguration); return awaits RequestAuthorization(); In this you have defined the service configuration, built the authorization request, and the intent to call as soon as the authorization is complete, and then wait for the authorization request. To build the authorization request is as follows: empty private BuildAuthorizationRequest(AuthorizationServiceConfigurationConfiguration service) { AuthorizationRequest.Builder = new AuthorizationRequest.Builder(serviceConfiguration, Configuration.ClientId, ResponseTypeValues.Code, Uri.Parse(Configuration.LoginRedirectUri)); constructor. SetScope (string. Join (Scopes)); authorizationRequest = constructor. Build(); } The job of this method is to abstract the Authorization Work.Constructor and create the request. Next, you need to construct the intent for once the operation is completed: Empty Private BuildCompletedIntent (AuthorizationServiceConfiguration serviceConfiguration) { Intent Intent = New Intent (Context, Type (MainActivity)); Intent. PutAuthStateExtra (new AuthState()); completedIntent = PendenteIntent.GetActivity (context, authorizationRequest.GetHashCode(), intent, 0); } The 'intent' you want to perform here is to return to your MainActivity with a new AuthState attached. Finally, in this flow is to handle the execution of the request: request for murdered tasks<AuthorizationResult> privateAuthorization() { authorizationService.PerformAuorizationRequest (authorizationRequest, completedIntent); wait taskCompletionSource.Task; return new AuthorizationsResult() { AccessToken = authorizationState?. AccessToken, IdentityToken = authorizationState?. IdToken, IsAuthorized = authorizationState?. Isauthorized?? false, RefreshToken = authorizationThis?. RefreshToken, Scope = AuthorizationThis?. Scope }; } Because the PerformAuthorizationRequest is synchronous and returns null, the code waits for the taskCompletionSource member, knowing that it will only be set when a is recovered. At this same point you know that the authorization state will be populated (if</AuthorizationResult> </AuthorizationResult> </AuthorizationResult> you can return your values as part of the Authorization Result. The second public NotifyCallback method, as I mentioned before, is what you want the MainActivity class to turn back on once your above-completed knowledge runs. In this method you want to check the response, update the state accordingly, and, if successful, perform a token exchange request: if (!intent. TryGetAuthStateFromExtra (out authorizationState)) { taskCompletionSource.SetResult(false); return; } Authorization Desresponse Response = AuthorizationResponse.FromIntent (intent); AuthorizationDesign exception = AuthorizationException.FromIntent (intent); authorizationState.Update(authorizationResponse, authorizationException); if (authorizationException != null) { taskCompletionSource.SetResult(false); return; } authorizationService.PerformToken(authorizationResponse.CreateTokenExchangeRequest(), ReceivedTokenResponse); Here you can see in the failure cases that I set the result of theCompletionSource task to false, and that it will unlock the Request Authorization method above. Additionally, the

PerformTokenRequest method receives a delegate, ReceivedTokenResponse, to run as soon as it is completed. This method is as follows: Private void ReceivedTokenResponse (TokenResponse tokenResponse, AuthorizationException authorizationException) { authorizationState.Update(tokenResponse, authorizationException); taskCompletionSource.SetResult(true); } At this point, you must have all the necessary authorization data and therefore can update the state accordingly (where you will find the values to return from the login method) and set the result to unlock theCompletionSource task. Implement authentication on your Xamarin interface As a cleanup, if you wish, feel free to remove all references to the 'Floating Action Bar' (also written as 'FAB') within the main files of the activity/axml class, as they are unnecessary bloated at this stage. To allow the user to sign in, you now need to implement this functionality in the user interface. Given the standard design, the most intuitive way to do this would be to add an item to the overflow menu in the upper right corner. You can do this by editing the menu_main.xml file in the Resources -> menu folder and adding this item as a child from the menu tag: <item android:id=@+id/action_signin android:orderincategory=100 android:title=@string/action_signin app:showasaction=never></item> With this code you have created a new option with a title to be set in the string feature file. As mentioned earlier in Android it is best practice to put any user facing text in the string feature file. To declare this data, edit the strings.xml file in the Resources -> values folder and add these lines: <string name=action_signin>Sign <string name=welcome_message_format>Hi, %1$s!</string> Not only did I declare a string for the 'Sign in' button here, but I also added above a sequence to a welcome message to the user once they have have In. The equivalent of the C# code of this string would be Hi, {0}! ', where the placeholder is string type. Note with all updates to these resource-based files, class Resource.designer.cs will automatically regenerate with new IDs for each object you have created, which can be referenced within your code. If this is not working for a particular file, select it in Solution Explorer and look at the Properties window. Make sure that the CustomTool property is set to the Value MSBuild:UpdateGeneratedFiles, as this is probably missing and preventing the designer file from recognizing it as a resource. Then add a ProgressBar to the existing activity_main.axml layout file: <ProgressBar android:id=@+id/signin_progress android:layout_width=wrap_content android:layout_height=wrap_content android:layout_gravity=center android:pbottomadding=12dp android:visibility=gone></ProgressBar> This ProgressBar (or spinner, as the case is), has an ID that you can reference with code and is configured to sit around the center of the screen. Visibility is set to go for now, but once your authorization code is running, you can set it to visible and inform the user that the application is busy. You now have a button to open authentication and a progress spinner to inform the user that the app is busy, it's time to use them. Within your MainActivity class, add the following property to the Activity attribute (above the class header): LaunchMode = LaunchMode.SingleTask This property ensures that there is only one instance of the MainActivity class, and you do not continue opening new ones. After you do this, add a static member variable to the Authorization Provder that you wrote above and create an instance of it within the existing replacement of the OnCreate method. Note that this must be done after the existing code within the method: authorization of static private authorization Provider; Empty Replacement Protected OnCreate (Bundle savedInstanceState) { ... authorizationProvider = new AuthorizationProvider(this); } Then override the OnNewIntent method. The purpose of this is when a new intent of this class is created (that is, when the external signal in the window returns), you call the NotifyCallback method from the AuthorizationProvider. Also included in this is a quick check to make sure that it is the expected flow: OnNewIntent protected replacement empty (Intent intent) { base. OnNewIntent (intent); if (intent != null & intention. Data.Path.Equals(/callback, StringComparison.OrdinalIgnoreCase) { authorizationProvider.NotifyCallback(intent); } } Now add the code behind the added menu item. In the existing replacement of the elected OnOptionsItemS method, add an if statement with a call to a new method that will handle the in process as follows: if (id == Resource.Id.action_signin) { OnSignInAttempted(); return true; } This new method will begin by making the ProgressBar that you added moments ago visible; to retrieve any component from the layout file, use the generic FindViewById method and insert the component component as your argument. After that, make a call to the SignInAsync method and wait for its result. Once the call has returned, the result is checked as authorized. If this authorization fails for any reason, an error dialog box appears, and the progress spinner disappears again. I'll fail to detail the success case for now, as you still need a place to go in this case: empty private async OnSignInAttempted() { ProgressBar signInProgress = FindViewById<ProgressBar>(Resource.Id.signin_progress); signInProgress.Visibility = ViewStates.Visible; AuthorizationResults of results = waits for authorizationProvider.SignInAsync(); if (!string. IsNullOrWhiteSpace (authorizationResult.AccessToken) && authorizationResult.IsAuthorized) { // Placeholder: Success case } else {/ Display an error for user AlertDialog authorizationErrorDialog = new AlertDialog.Builder(this) . SetTitle (Error!) . SetMessage (We can't authorize you.) . Create(); authorizationErrorDialog.Show(); signInProgress.Visibility = ViewStates.Gone; } } When the user is authenticated, you must redirect them to the next page of their experience. If you remember previously, each page is represented by an Activity, which is why you need to create a new one now. To start, within the 'Resources -> layout' folder you will need to create the new activity layout file activity_dashboard.axml. The easiest way to do this is by going to the New Item... option in the context menu and selection of the 'Android Layout' template. Inside your new layout file add a <LinearLayout xmlns:android= android:orientation=vertical android:layout_width=match_parent android:layout_height=match_parent android:gravity=center> <TextView android:id=@+id/welcome_message android:textappearance=?android:attr/textAppearanceMedium android:layout_width=wrap_content android:layout_height=wrap_content android:layout_centerinparent=true></TextView> </LinearLayout> Simple TextView component to display text like this: In this excerpt, you have a TextView component with a reference ID centered in the middle of the page, from which you will display a welcome message. Then create a corresponding 'DashboardActivity' activity class through the 'New Item...' option in the project context menu in the solution explorer and selecting the 'Activity' template. To link this class to your layout file, you need to call the SetContentView function in the generated Method OnCreate() (under the invocation of the legacy base method): SetContentView(Resource.Layout.activity_dashboard); To customize your welcome message, you'll want to pass the user name to your new activity. If you remember previously, the best way to do this was with extras about intent, and you created a class of extensions to handle it. As before, add new to 'Put' and 'Get' an extra 'name' in the file IntentExtensions.cs you created above: private const string NameKey = Name; public static void </ProgressBar> </ProgressBar> Intent, string name) { intent. PutExtra (NameKey, name); } GetNameExtra public static sequence (this intent of intent) { return intent. GetStringExtra(NameKey); } Now, using this extended functionality, after the call to SetContentView you made in the OnCreate() method, retrieve the user name and set the text of the TextView component appropriately: sequence name = Intent.GetNameExtra(); TextView receivesMessage = FindViewById<TextView>(Resource.Id.welcome_message); welcomeMessage.Text = Resources.GetString(Resource.String.welcome_message_format, name); In this extract, when retrieving the TextView instance, its value is set for its welcome message, from which it is created using the equivalent of string android features. Format(). With this, your activity on the dashboard is complete, and now you need to call it. In the placeholder for the success case I left open from the OnSignInAttempted method, you can achieve this by adding the following code: JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler(); JwtSecurityToken jsonToken = tokenHandler.ReadJwtToken (authorizationResult.IdentityToken); Claims yenable<Claim> = jsonToken?. Load?. Claims; string name = affirmations?. FirstOrDefault(x => x.Type == name)?. Value; Intent Dashboard = new Intent (this, typeof(DashboardActivity)); dashboardIntent.PutNameExtra (name); StartActivity (dashboardIntent); Finishing(); The first block reads the token and retrieves the user's name (if it exists). The second that a new intent is created for dashboard activity, the user name is stored in this intent using its extension method set above, and then the activity starts (that is, navigated). To prevent the user from navigateback to this page later, the code ends by calling the Finish() method. Run your Android app Now it's time to launch your app using your chosen device! If you are debugging using the emulator, this should be as simple as hitting F5, which will first open and load the emulator, and then the code will be deployed to it. As a side note, you don't need to close the emulator between execution/debugging attempts, because it only needs to redeploy the application. If you are debugging using a device that has not been used for this purpose before, you will need to configure the device first. Do this by enabling developer options and within the new menu by linking 'Android debugging' under the header 'Debugging'. After that, simply plug the device in, accept the dialog box on your device confirming that this is a secure debugging connection, and you should be able to deploy and run your application with F5. Note that physical devices take precedence higher than the emulator and will switch to it as the default debug option when connected. Since your has been deployed and loaded, you will be greeted by the standard single-page template. Open the menu in the upper right corner to log in, and once you have entered your data, you must return to this page with the progress bar spinning before being automatically sent to the panel page</Claim> </TextView> </TextView> Your welcome message: Learn more about Xamarin, OpenID Connect, and Secure Authentication If you followed along with all these steps, you now have a basic Android app built using Xamarin.Android, with fully functional user authentication based on OpenID and okta service. From here, you can easily expand control panel activity to implement its functionality. To see the code of this full-headed post for our GitHub page. If this tutorial has sharpened your appetite for the development of Xamarin and you would like to learn more, then I strongly suggest you take a look at these other great articles: As always if you have any questions, comments or concerns about this post feel free to leave a comment below. For other great content from Team Okta Dev, follow us on Twitter @OktaDev, Facebook, LinkedIn and YouTube! Youtube!