Windows

Windows apps concept mapping for Android and iOS developers

Version 1.0
04 / 20 / 16

MICROSOFT CORPORATION

Microsoft

Introduction

This document defines how fundamental software development concepts map across to Android, iOS and Windows. Developers new to Windows and familiar with Android or iOS will be able to use this reference guide to understand how to work with Windows. The guide will include a master table which draws out the relationship between each pair.

The table rows will look like this:

Fundamental software concept.

As found in Android.

As found in iOS.

As found in Windows.

Links to additional resources:

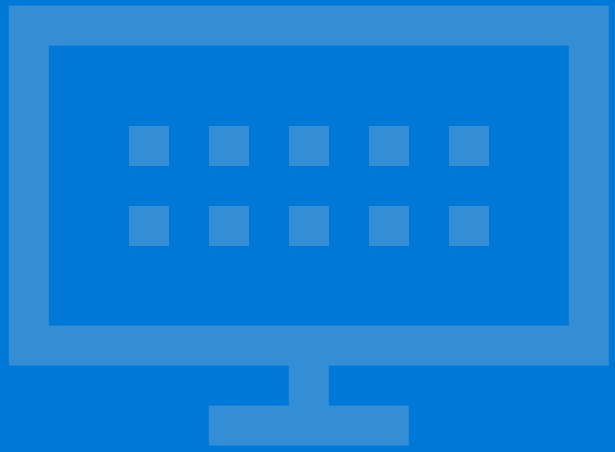
Developing apps for Windows

Windows apps concept mapping guide on MSDN

Downloadable PDF of this whitepaper

Windows developer Virtual Machines (VM)

UI



Design language.
A set of conventions that prescribe how apps on the platform should look and behave.

Android Material Design guidelines provide a visual language for Android designers and developers to follow.

Human Interface Guidelines provide advice for iOS designers and developers.

UWP Windows Apps Design shows you how to create an app that looks fantastic on all Windows 10 devices. You will find UI design fundamentals, responsive design techniques, and a full list of detailed guidelines.

User interface markup language.
A markup language that renders and describes a UI and its components. Each platform provides an editor for both visual and markup editing.

XML layouts, edited using **Android Studio** or **Eclipse**.

XIB and **Storyboards** edited using **Interface Builder** inside Xcode.

XAML, edited using **Microsoft Visual Studio** and **Blend for Visual Studio**.

XAML platform

Create a UI with XAML

Define Layouts with XAML

Built-in user interface controls.
Reusable UI elements provided by the platform such as buttons, list controls, and text controls.

Prebuilt **view** and **view group** classes referred to as widgets, layouts, text fields, containers, date/time controls and expert controls.

Views and **controls** found in the Xcode object library and listed in the UIKit user interface catalog. Views include image views, picker views and scroll views. Controls include buttons, date pickers and text fields.

The XAML platform provides you with a generous set of **built-in controls** such as buttons, list controls, panels, text controls, command bars, pickers, media, and inking.

Add controls and handle events

Control event-handling.
Defining the logic that runs when events are triggered within UI controls.

Event handlers and **event listeners** are added in XML or programmatically.

Controls send **action** messages to **targets**.

You can define methods to handle the events of a XAML control in a **code-behind file** attached to the XAML page. **Event handlers** are always written in code. But you can hook those handlers to events either in XAML or in code.

Add controls and handle events

Events and routed events overview

Data binding.
A software design pattern that allows your app UI to render data and optionally stay in sync with that data.

There is a **Data Binding Library** provided, although it is still in beta.

No built-in bindings system exists on iOS. **Key-value observing** can be built upon to perform data binding, either with the use of a third-party library, or writing additional code. Controls use a delegate/callback approach for obtaining data.

The UWP platform handles data binding for you. You use the **{x:Bind}** markup extension to take advantage of high performance binding or **{Binding}** to take advantage of more features. It's then just a case of configuring your binding to choose whether the platform uses **one-way binding** to display values from a data source in your UI, or whether it also observes those values and updates your UI when they change with **two-way binding**.

Data Binding

UI Automation.
Programmatic access to UI elements, making apps accessible to assistive technology products and enabling automated test scripts to interact with your UI.

Text labels, **contentDescription** and **hint** values help ensure UI elements can be found by automation. Android Studio allows you to write UI tests using the **UI Automator** and **Espresso** testing frameworks.

The **Automation instrument** allows you to write automated UI test scripts which identify elements using the **accessibility** settings or the element's position in the **element hierarchy**.

You get programmatic access to built-in UI elements in UWP out-of-box with **UI Automation**.

Custom Automation Peers allow you to provide automation support for your own custom UI classes. The **Coded UI Test Project** in Visual Studio allows you to automatically test your whole application through the UI, or to test the UI in isolation.

Changing the appearance of a control.
Editing size, color and other attributes.

Controls have **properties** which can be edited using the designer tool, in XML markup or programmatically.

Controls have **attributes** which you can edit using the **Attributes Inspector** in Interface Builder or programmatically.

You can edit the **properties** of controls in the XAML markup or programmatically, using Visual Studio and Blend for Visual Studio.

Add controls and handle events

Reusable visual styles.
Apply visual changes to a number of controls, in a reusable format.

XML styles are sets of properties that are applied to one or more controls.

iOS does not support reusable visual styles out-of-box, but the UIAppearance protocol allows multiple controls to share common attributes.

You can create reusable **styles**, which can be applied to multiple controls and stored in a **ResourceDictionary** for easy reuse.

Quickstart: Styling Controls

Editing the visual structure of controls.
Customize the visual structure of a control beyond just modifying properties or attributes, e.g. moving the checkbox text underneath the checkbox.

No simple method of editing the visual structure of controls exists in Android.

No simple method of editing the visual structure of controls exists in iOS.

To customize the visual structure of a control, you can copy and edit its **control template** in XAML markup.

Quickstart: Control Templates

Built-in touch gestures.
Provide customized touch support by handling high level abstracted gesture events such as tap and double tap in views and controls.

Gesture detectors detect common touch gestures including scrolling, long-press, tap, double-tap and fling.

UIKit framework provides built-in **gesture recognizers** which detect touch gestures including tap, pinch, pan, swipe, rotate and long-press.

UI elements allow you to handle **static gesture events** including tap, double-tap, right-tap and holding, as well as **manipulation gesture events** including slide, swipe, turn, pinch and stretch. Gesture events are **routed events** and can be handled by parent objects containing the child UIElement.

Touch interactions

Custom user interactions - gestures, manipulations, and interactions

Navigation and app structure

Layouts.
The layout defines the structure of the user interface.

Layout is composed of **view groups** such as the **LinearLayout** and the **RelativeLayout** which can nest other view groups or views.

Layout is composed of a **UIViewController** containing **UIView's** which can be nested.

XAML which provides a flexible layout system composed of **layout panel classes** such as **Canvas**, **Grid**, **RelativePanel** and **StackPanel** for static and responsive layouts. **Properties** are used to control the size and position of the elements.

Define layouts with XAML

Peer-to-peer navigation.
Presenting the user with methods of navigating between pages of equal hierarchical importance.

Tabs, **swipe views** and **navigation drawers** provide **lateral navigation**.

Tab bar controllers, **split view controllers** and **page view controllers** allow navigation between views of equal hierarchy.

You can display a persistent list of links/tabs above the content using **tabs/pivots**. The **navigation pane/ split view** lets you display a list of links alongside the content.

Navigation

Peer to peer navigation between two pages

Hierarchical navigation.
Navigating between parent and child pages of a hierarchy.

Lists, and **grid lists**, **buttons** and other controls provide **descendent navigation** when used with **intents** to load other **activities**.

Navigation controllers allow users to navigate between levels of a hierarchy.

Hubs let you show the user a preview of content which can be selected to navigate to child pages. **Master/ details** let users pick from a list of item summaries which display next to the corresponding detail section.

Navigation

Back button navigation.
Navigating back through an application.

The **back** and **up** buttons inside the action bar provide **ancestral** and **temporal** navigation using the **back stack**.

The **navigation controller** can have a back button added to it.

You can handle software or hardware back button presses easily using the **back stack property** which allows your users to traverse the **navigation history**.

Back button navigation

Splash screen.
Showing an image on app launch, used primarily for branding.

Splash screens are not provided by default, and are implemented by editing the first activities **theme background**.

Apps must either have a **static launch image** or **XIB/storyboard launch file**.

You create a splash screen using an **image** and a colored background. Splash screen time can be extended.

Add a splash screen

Guidelines for splash screens

Custom inputs

Voice.
Speech recognition for speech input, and additional voice capabilities.

Speech input can be provided by any app which implements a **RecognizerIntent**, such as **Google Voice Search**. The **SpeechRecognizer** class allows apps to use Google's speech recognition API.

No built-in speech recognition or speech input APIs exist.

You can use the **speech recognition** API to interact with your app in the foreground. You can use speech-based **Cortana interactions** to launch apps in the foreground or background, and to interact with background apps.

Speech interactions

Custom user inputs.
Handling keyboard, mouse, stylus and other inputs.

Support for interactions includes **touch**, **touchpad**, **stylus**, **mouse** and **keyboard**. Movements and inputs are reported in the same way as touch, but it is possible to detect more information about the **input device**.

Support for **touch**, the **Apple Pencil** and hardware **keyboards** are provided.

You will find support for a wide range of interactions including **touch**, **touchpad**, **pen/stylus** with digital ink, **mouse** and **keyboard**. Your apps can handle the data without needing to know which input device was used, and raw input device data can be accessed if needed.

Handle pointer input

Custom user interactions

Data

Local app data.
Storing settings and files related to your app locally.

Local files can be saved using **openFileOutput** and **openFileInput**. Settings in a **shared preferences file** can be accessed using **getSharedPreferences**.

Local files can be stored in the **application support** directory, accessed via the **NSFileManager** class. Settings in **preferences** files can be accessed by the **NSUserDefaults** class.

The **Windows.Storage** classes handle local data storage for you in a unified way. You store settings as an **ApplicationDataContainer** object, accessed via the **ApplicationData.LocalSettings** property. You store files in a **StorageFolder** object accessed via the **ApplicationData.LocalFolder** property.

Store and retrieve settings and other app data

Local database storage.
Storing app data in a relational database, with object-relational mappers (ORM) if applicable.

The **SQLite** database is provided. No ORM is built-in. SQL queries are run using the **SQLiteDatabase** class.

The **SQLite** database is provided. **CoreData** is the built-in object graph framework which can be used with SQLite and provide functionality comparable with an ORM.

You can store data using **SQLite**. **Entity Framework** is a built-in ORM which eliminates the need to write lots of data access code and enables you to easily query the database without writing SQL. You can run SQL queries directly with the SQLite library.

Data Access

HTTP libraries for REST access.
Built-in libraries that let you communicate with web services and web servers using HTTP(S).

HTTP libraries **HttpURLConnection** and **Volley**.

NSURLSession, **NSURLConnection** and **NSURLDownload**.

You can use the built-in **HttpClient** API to access common HTTP functionality including GET, DELETE, PUT, POST, common authentication patterns, SSL, cookies and progress info.

Cloud backup services.
Platform-provided backup services for app data.

Android's **backup manager** handles the backing up of application data in Google's **Android Backup Service**.

iCloud Backup can be configured by a user to handle their backups, including app data. Apps which use iCloud compatible **Core Data**, the **iCloud key-value store** and **iCloud document storage**.

Any app data that you store using the roaming **ApplicationData APIs** (including **RoamingFolder** and **RoamingSettings**) will be automatically synced to the cloud and to the user's other devices, too. The syncing is done by way of the user's Microsoft account.

Guidelines for roaming app data

HTTP file downloads.
Downloading large and small files over HTTP.

URLConnection and **HTTPURLConnection** are used to download over HTTP and FTP, it is also possible to make use of the system **download manager** to download in the background.

NSURLSession and **NSURLConnection** can be used to download files over HTTP and FTP.

The **background transfer API** lets you reliably transfer files over HTTP(S) and FTP, taking into account app suspension, connectivity loss and adjusting based on connectivity and battery life. You can also use **HttpClient** which is ideal for smaller files.

Which networking technology?

Background transfers

Sockets.
Creating low level UDP datagram and TCP sockets to communicate with other devices using your own protocol.

Socket class provides TCP sockets, **DatagramSocket** class provides a UDP socket.

NSStream and **CFStream** provide TCP sockets, **CFSocket** provides UDP sockets.

You can use the **DatagramSocket** class to communicate using a UDP datagram socket and the **StreamSocket** class to communicate over TCP or Bluetooth RFCOMM.

Networking basics

Which networking technology?

Sockets overview

WebSockets.
Provide two-way communication between a client and server, enabling real-time data transfer.

No built-in WebSockets libraries exist on Android.

No built-in WebSockets libraries exist on iOS.

Secure connections to servers supporting WebSockets can be made with the **MessageWebSocket** class for smaller messages with receipt notifications and **StreamWebSocket** for larger binary file transfers which can be read in sections.

Networking basics

Which networking technology?

WebSockets overview

OAuth libraries.
OAuth libraries allowing access to third party OAuth providers, and any account management built into the platform.

No generic OAuth library is provided. The **GoogleAuthUtil** class is provided for OAuth authentication with Google Play Services.

No generic OAuth library is provided. The **accounts framework** provides access to user accounts already stored on the device such as Facebook and Twitter.

The generic OAuth library **Web authentication broker** lets you connect to third-party identity provider services. The **Credential locker** allows your users to save their login and use it on multiple devices. The **Microsoft.Live** namespace lets you easily access Live SDK OAuth for access to Microsoft services.

Authentication and user identity

Windows.Security.Authentication.Web API documentation

WebAuthenticationBroker code example

Tooling

IDE.
The toolset used to create your app.

Android Studio and **Eclipse**, with Google pushing developers toward the use of Android Studio.

Xcode

Visual Studio and **Blend for Visual Studio** has all the tools you need to code, design, connect, debug, analyze, optimize and test UWP apps. Visual Studio also provides you with **emulators** for Windows 10 devices, so you can test your app across a range of emulated devices.

Downloads and tools for UWP

Code organization.
The basic folder structure of an app, often created from an initial template.

AndroidManifest file, **java** folder containing source files, **res** folder with resources including layouts and values, **Gradle** build scripts in Android Studio and **Ant** build scripts in Eclipse.

Source files and **Supporting Files**, **Info.plist** file, **Main.storyboard** and **LaunchScreen.storyboard**. Images are stored in **Asset libraries**.

Your UWP app contains XAML and code files for your app, various images in the **Assets folder**, a start page such as **MainPage.xaml** and **MainPage.xaml.cs** and a manifest.

Create a hello world app

App lifecycle

App lifecycle.
Handling events on app launch, suspension, resume and close, providing an opportunity to save/ restore application state and run other tasks.

Each activity has its own **activity lifecycle** with states such as **resumed**. **Lifecycle callbacks** such as **onResume** are implemented in in your **activity classes**.

The **application lifecycle** has states such as **suspended**. Methods such as **applicationDidEnterBackground**: are implemented in the **application delegate object** to run code on state changes.

Your application has the **app execution states** NotRunning, Activated, Running, Suspending, Suspended and Resuming.

You can implement the **Application** class methods OnLaunched, OnActivated, Suspending or Resuming in your app to run code when the state changes.

App lifecycle

Background tasks.
Tasks that perform background operations and continue to run when the app is no longer in the foreground.

Apps can launch **services** which perform background operations when the app is no longer in the foreground. Services have their own **lifecycle** and are registered in the manifest.

Background execution is only permitted for specific task types.

Apps declare **supported background tasks** in the Info.plist file using the **UIBackgroundModes**.

The system controls when background tasks are run and for how long.

You can create a background task by implementing the **IBackgroundTask** interface and registering the task in the application manifest. You can set a task to be triggered with a **timer**, **system trigger**, or **maintenance trigger**.

Support your app with background tasks

Create and register a background task

Guidelines for background tasks

Performance

Performance best practices.
Guidelines for building apps that are fast, responsive, considerate of battery life with a fast startup time.

Android provides the **Best Practices for Performance** training guide.

iOS provides the **Performance Overview** document.

You can read the detailed **Performance guide** with sections covering topics such as: setting performance goals, measuring performance, memory management, smooth animations, efficient file system access and the tools available for profiling and performance.

View optimization for a responsive UI.
Improving performance by optimizing views.

Optimizing **layout hierarchies** using the Hierarchy Viewer tool, **reusing layouts** and loading **views on demand** are all techniques to help keep the UI thread responsive and avoid "Application Not Responding" dialogs (**ANR's**).

Fixing UI issues with **offscreen rendering**, **blended layers**, **rasterization** using the **Core Animation** tool help keep the UI thread responsive.

You can easily **optimize** XAML **markup** and **layouts** by following a few simple steps. Techniques include reducing layout structure, minimizing the element count and minimizing overdrawing.

Keep the UI thread responsive

Optimize your XAML markup

Optimize your XAML layout

Threading.
Use of threading to maintain a **responsive UI** and run multiple **tasks in parallel**.

Threading is achieved using the classes **Runnable**, **Handler**, **ThreadPoolExecutor**, and the higher level **AsyncTask**.

Threading is achieved using **NSThread**, **Grand Central Dispatch**, and the higher level **NSOperation**.

You can work with threads by submitting **work items** to the **threadpool** with **RunAsync**. You can use a timer to submit a work item with **CreateTimer** and create a repeating work item with **CreatePeriodicTimer**.

Submit a work item to the thread pool

Use a timer to submit a work item

Create a periodic work item

Best practices for using the thread pool

Asynchronous programming.
Avoid threading complexity by taking advantage of asynchronous programming patterns to keep the UI thread responsive.

The use of **threading** is required to create your own asynchronous classes. Some built-in classes are asynchronous.

The use of **threading** is required to create your own asynchronous classes. Some built-in classes are asynchronous.

You can use asynchronous patterns to avoid blocking the main thread when you create your own APIs, e.g. using **async** and **await** in C# and Visual Basic. You can use the asynchronous built-in APIs which end in the word **Async**.

Asynchronous programming

Call asynchronous APIs in C# or Visual Basic

List view optimization.
Built-in patterns to aid with optimizing lists of data, which often have poor performance when large amounts of data need to be shown.

The **ViewHolder** design pattern is used to avoid multiple view lookups, which allows you to use reusable UI elements.

A range of optimizations can be made to improve the performance of **UITableView**, nothing is built-in.

You can use the ListView and GridView controls which provide **UI virtualization** out-of-box, providing a smooth panning and scrolling experience and a faster startup time. You can also implement IList and INotifyCollectionChanged in your data source, providing **data virtualization** and further improving performance.

ListView and GridView UI optimization

ListView and GridView data virtualization

Monetization

In-app purchases.
Platform features that allow users to make purchases in your apps.

In-app billing is provided by Google Services. Products are added to the **Google Play Developer Console**. In-app purchases are implemented with the **Google Play Billing Library**.

Products are added to **iTunes Connect**. In-app purchases are implemented using the **StoreKit** framework.

Products are purchased using **SKMutablePayment** and **SKPaymentQueue**.

You create in-app product purchases for your app by adding them to your app and submitting them to the Store.

You use the **CurrentApp** class to define in-app purchases.

You use **CurrentApp.RequestProductPurchaseAsync** to display the UI that allows customers to purchase the product.

Enable in-app product purchases

Consumable in-app purchases.
In-app products which can be purchased, used and then purchased again.

Consumable purchases are enabled by making a regular purchase and then consuming it with **consumePurchase**, enabling it to be purchased, used, and then purchased again.

Consumable products are **defined as consumable products** in iTunes Connect.

You can support consumables by defining their product type as Consumable when you submit them to the Store. You then call **CurrentApp.ReportConsumableFulfillmentAsync** after a consumable purchase has been made to allow the customer to access it.

Enable consumable in-app purchases

Testing in-app purchases.
Enabling you to test your in-app purchase code without putting your app in the Store.

The **in-app billing sandbox** is used for testing.

Sandbox tester accounts are used for testing.

You can test in-app purchases by simply using the **CurrentAppSimulator** class in place of CurrentApp.

Trials.
Enabling you to easily limit content or include advertising based on a trial version of an app.

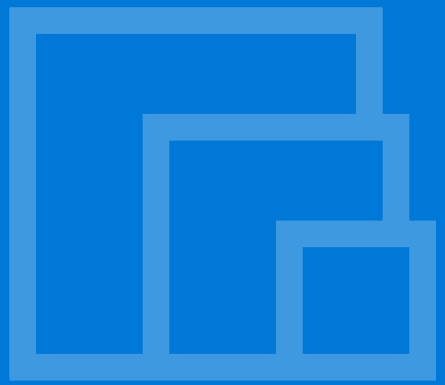
Google Play **doesn't officially support app trials**. Trials or including advertising is achieved by creating an in-app purchase and taking the appropriate code path when confirming the purchase was successful.

The App Store **doesn't officially support app trials**. Trials or including advertising is achieved by creating an in-app purchase and taking the appropriate code path when confirming the purchase was successful.

You can offer a free trial version of your app by using the **'Free Trial' option** when submitting your app to the Store. You can then use **LicenseInformation.IsTrial** to check the trial status of the app and present different code paths accordingly. You can register for the LicenseChanged event to be notified when the user changes the trial status while the app is running.

Exclude or limit features in a trial version

Building for diversity



Adaptive UI: flexible layouts.
Supporting different screen sizes with a flexible height and width.

Flexible layouts can be achieved using the **wrap_content** and **match_parent** values in LinearLayout objects, or by making use of RelativeLayout objects for alignment.

Flexible layouts can be achieved using the **adaptive model** with universal Storyboards, making use of **Auto Layout** with **constraints** and **traits** such as horizontalSizeClass and displayScale which are applied to view controllers.

You can create a fluid layout using **layout properties** and **panels** with a combination of fixed and dynamic sizing.

Define layouts with XAML - layout properties and panels

Responsive design 101

Adaptive UI: tailored layouts.
Supporting different screen sizes with separate targeted layouts.

Providing alternative layout files for different screen configurations in the resources directory using **configuration qualifiers** such as **small**, **large**, **ldpi**, and **hdpi** allows you to target custom layouts to screens of varying size and density.

Define a **separate iPhone and iPad Storyboard** to tailor layouts to different device families in a universal app.

You can build a tailored layout by defining **different XAML markup files** per device family.

Define layouts with XAML - tailored layouts

Adaptive UI: responsive layouts.
Responding to changes in screen size, such as rotation, or a change in the size of a window.

Use of flexible layouts with **LinearLayout** and **RelativeLayout**, or providing alternative layout files for different orientations enable responsive layouts.

When the **size** or **traits** of a view change, the **constraints** specified in storyboards are applied.

You can easily reflow, reposition, resize, reveal, or replace sections of your UI at runtime in response to window size changes using **VisualState**, the **VisualStateManager** and **AdaptiveTrigger**.

Define layouts with XAML - visual states and state triggers

Responsive design 101

Supporting different device capabilities. Take advantage of advanced hardware features while still supporting devices without them.

Testing for device features at runtime using **PackageManager.hasSystemFeature** enables you to decide if hardware specific code can run.

There is **no single check** you can perform at runtime to test for device features, you test for each feature in a specific way to decide if hardware specific code can be run.

You can add **platform extension SDKs** to your package to target additional functionality found in different device families including phone, desktop, and IoT. You use the **ApiInformation API** to test for the presence of types and members at runtime, and can call those types and members only if they're present.

Supporting different device capabilities. Take advantage of advanced hardware features while still supporting devices without them.

The **Android Support Library** can be packaged with your app to make some newer APIs available to those with older versions of Android. Testing for the API level at runtime can be done using **Build.Version.SDK_INT**.

Standard runtime checks are used to find out if APIs are available, such as the **class** method to check if a class exists and **respondsToSelector:** to check for methods on classes.

You can use **ApiInformation.IsApiContractPresent** to identify if an API contract with a specified major and minor number is present. You also use the **ApiInformation API** to test for the presence of types and members at runtime, and can call those types and members only if they're present.

Notifications

Tiles and badges.
Present updates to users on the home screen.

App Widgets are views on your application that can be embedded into the home screen and can receive periodic updates. **No badge system** exists on Android. No identical system to tiles exists.

No tiles or widgets exist on iOS. You can add a **badge** to your icon with a number which can change in response to local or remote notifications.

Your app has a **tile** which can be pinned to the start screen and is used to display your choice of text, images, and a **badge** with glyphs and numbers. You can update the content of tiles from the app via push notifications or at predefined schedules. Tiles can be adaptive, and can change according to where they are being displayed.

Create tiles

Create adaptive tiles

Choose a notification delivery method

Guidelines for tiles and badges

Displaying notifications.
Types of notifications that can be displayed.

Notifications can be shown in the **notification area** and **notification drawer**, **heads-up notifications** present a notification in a small floating window. Notifications can have actions added to them by defining a **PendingIntent**.

Pop-up notifications appear as **banners** or **alerts**. You can add custom action buttons to **actionable notifications** which are defined with **UIMutableUserNotificationAction**.

You can create adaptive pop-up notifications called **toast notifications**. You can define toasts in XML with visual content, **actions** which can be buttons, or inputs and audio.

Adaptive and interactive toast notifications

Choose a notification delivery method

Guidelines for toast notifications

Scheduling local notifications.
Local notifications sent by your app at a scheduled time.

Notifications and actions are defined using a **NotificationCompat.Builder** and can be scheduled and handled in-app using **AlarmManager** and **BroadcastReceiver**.

Local notifications are created using **UILocalNotification**, and can be scheduled with **UILocalNotification: scheduleLocalNotification**.

You can schedule a toast notification using **ScheduledToastNotification**. You can send a tile notification from your app using the **TileNotification** class, or schedule a tile notification with ScheduledTileNotification.

Adaptive and interactive toast notifications

Send a local tile notification

Sending push notifications.
A notification sent from a push notification server and optionally handled in-app.

Google Cloud Messaging provides push notification support for Android.

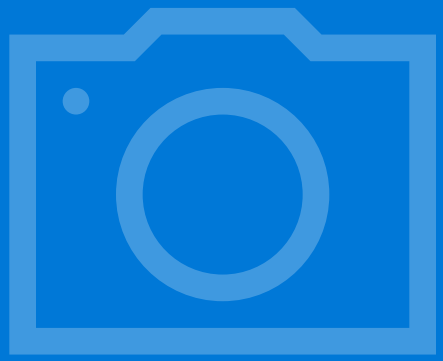
Remote or push notifications are provided by the **Apple Push Notification service (APNs)**.

You receive push notifications sent from **Windows Push Notification Services (WNS)** which can be of type tile, toast, badge, or raw notification. You can use the PushNotificationReceived notification delivery event to receive notifications while the app is running.

Windows Push Notification Services (WNS) overview

Raw notification overview

Media capture and rendering



Capturing media.
Recording audio and visual content.

Using an **intent** such as MediaStore. ACTION_VIDEO_CAPTURE allows media to be captured with an existing camera app. Using the **android.hardware.camera2** or **camera** library enables the implementation of a custom camera interface. **MediaRecorder** APIs can be used to capture audio.

The **UIImagePickerController** allows for the capture of video and photos with the system UI. The **AVFoundation** classes such as **AVCaptureSession** enable direct access to the camera. The **AVAudioRecorder** class enables audio recording.

You can capture photos and video while using the built-in camera UI with the **CameraCaptureUI** class. You can interact with the camera at a low level, and capture audio with classes in **Windows.Media.Capture** such as the **MediaCapture API**.

Capture photos and video with CameraCaptureUI

Capture photos and video with MediaCapture

Media playback.
Playing audio and video files.

The **MediaPlayer** and **AudioManager** classes are used to play audio and video files.

The **AVKit framework**, **AVAudioPlayer**, and **Media Player Framework** are used to play audio and video files.

You can use the **MediaSource** class, **MediaElement**, and **MediaPlayer** classes to play back audio and video from sources such as local and remote files.

Media playback with MediaSource

Editing media.
Composing new media files from existing recordings and applying special effects.

Low level classes such as **MediaCodec**, **MediaMuxer**, and **android.media.effect** can be used for content editing.

Classes in the **AV Foundation** framework such as **AVMutableComposition**, **AVMutableVideoComposition**, and **AVMutableAudioMix** can be used for content editing.

You can use the **Windows.Media.Editing** APIs such as **MediaComposition** and **MediaClip** to create media compositions from audio and video files. You are able to add video and image overlays, combine video clips, add background audio, and apply audio and video effects.

Media compositions and editing

Sensors

Sensors.
Detect device movement, position and environmental properties.

The **sensor framework** is used to access hardware and software sensors with classes such as **SensorManager** and **SensorEvent**.

The **Core Motion framework** is used to access raw and processed sensor data.

You can use classes in **Windows.Devices.Sensors** to access sensor readings and events triggered when new reading data is received from the sensor.

Sensors

Location and mapping

Location.
Finding the device's **current** location and tracking **changes**.

The Google Play services location APIs provide high-level access to the **last known location** with the **fused location provider** using the **getLastLocation** and **requestLocationUpdates** methods. Low-level access is provided in the Android libraries with the **LocationManager**.

The **Core Location CLLocationManager** class is used to monitor a device's location, with **startUpdatingLocation** for the standard location service and **startMonitoringSignificantLocationChanges**for the **significant-change** location service.

You can track device location with classes in **Windows.Devices.Geolocation**. Use **Geolocator.GetGeopositionAsync** for a one-time reading. Use **Geolocator.PositionChanged** to obtain the location regularly using a timer, or be informed when the location has changed.

Get the user's location

Displaying maps.
Displaying an **interactive built-in map** and adding **points of interest**.

The **GoogleMap**, **MapFragment**, and **MapView** classes within the **Google Maps Android API** allow maps to be embedded in apps. Points of interest can be displayed using **markers** and the customizable **Marker** class.

Maps are embedded into iOS apps with the **MKMapView** class in the **MapKit framework**. **Annotations** can be added to apps to display points of interest using object classes such as **MKPointAnnotation** and view classes such as **MKPinAnnotationView**.

You can embed maps in your apps using the built-in **MapControl** XAML control which provides 2D, 3D, and streetside views. You can add points of interest with a pushpin, image, or shape using classes such as **MapIcon**, **MapPolygon** and **MapPolyline**.

Display maps with 2D, 3D, and Streetside views

Display points of interest (POI) on a map

Geofencing.
Monitor the entering and leaving of a particular geographic region.

Geofences are monitored using the **Location Services** in the Google Play Services SDK.

Regions are monitored with the **CLCircularRegion** class and registered with the **CLLocationManager.startMonitoringForRegion:**.

You can create a geofence with the **Geofence** class and define your **monitored states** such as entering or leaving a region. Handle geofence events in the foreground with the **GeofenceMonitor** class, and in the background with the **LocationTrigger background** class.

Set up a geofence

Geocoding and reverse geocoding. Converting addresses to geographic locations (geocoding) and converting geographic locations to addresses (reverse geocoding).

The **Geocoder** class is used for geocoding and reverse geocoding.

The **CLGeocoder** class is used for geocoding.

You can perform geocoding using the **MapLocationFinder** class in **Windows.Services.Maps**. You use **FindLocationsAsync** for geocoding and **FindLocationsAtAsync** for reverse geocoding.

Perform geocoding and reverse geocoding

Routes and directions. Providing routes, distances, and directions between two geographical locations.

Google provides the web service **Google Maps Directions API** which can be used on Android although no SDK is provided.

Map Kit provides the **MKDirections** API which can be used to fetch information about a route and directions.

You can request a walking or driving route with the **MapRouteFinder** class in **Windows.Services.Maps**. Routes are returned as a **MapRoute** instance which can be easily shown on a MapControl. Directions are returned inside the **MapRouteManeuver** object.

Display routes and directions on a map

App-to-app communication

Invoking another app.
Launching another app, and optionally sharing data such as links, text, photos, videos, and files.

An **implicit intent** is used to launch another app, by defining an **action** and optional data in an **Intent** and calling it with **startActivityForResult**.

App extensions can be used to provide access to app data to another app. **URL schemes** enable a URL to be passed to another app.

You can launch another app which has registered for a URI with **Launcher.LaunchUriAsync**, or **Launcher.LaunchUriForResultsAsync** to launch for results and get data back from the launched app. You can use **Launcher.LaunchFileAsync** to pass a file to another app to handle.

You can use a **share contract** to easily share data between apps.

Launch the default app for a URI

Launch an app for results

Launch the default app for a file

Share data

Allowing your app to be invoked. Allow your app to respond to a request from another app.

Apps register an **intent handling activity** with an **intent filter** to respond to an implicit intent from another app.

Packaging an **app extension** enables data to be shared with other apps. Apps can register a **custom URL scheme** using the **CFBundleURLTypes** key in Info.plist.

You can register your app to be the default handler for a **URI scheme name** by registering a **protocol** in the package manifest and updating the **Application.OnActivated** event handler, optionally returning results. In the same way you can register your app to be the default handler for certain file types by adding a declaration in the package manifest and handling the **Application.OnFileActivated** event.

You can handle share contract requests by registering your app as a share target in the manifest and handling the **Application.OnShareTargetActivated** event.

Launch an app for results

Handle file activation

Receive data

Copy and paste.
Copy and pasting text and other content between apps.

The **clipboard framework** can be used to implement copy and paste with the **ClipboardManager** and **ClipData** classes.

The **UIPasteboard**, **UIMenuController**, and **UIResponderStandardEditActions** can be used to implement copy and paste.

Many default XAML controls already support copy and paste. You can implement copy and paste yourself using the **DataPackage** and **Clipboard** classes in **Windows.ApplicationModel.DataTransfer**.

Copy and paste

Drag and drop.
Dragging and dropping content between apps.

Drag and drop can be implemented within a single application by using the **Android drag/drop framework**.

No high-level drag and drop APIs are provided by iOS.

You can implement dragging and dropping in your app to enable app-to-app, desktop-to-app, and app-to-desktop drag and drop capabilities. You implement drag and drop support in the UIElement class with the **AllowDrop**, and **CanDrag** properties, and the **DragOver**, and **Drop** events.

Drag and drop

Software design

Software design patterns.
Recommended or well-used patterns for the platform.

No formal pattern has been recommended or provided for Android development, although the beta Data Binding Framework may enable more widespread use of the **Model-View-ViewModel (MVVM)** pattern. A number of third party articles and frameworks recommend the **Model-View-Presenter (MVP)** and **MVVM** approaches.

Model-View-Controller (MVC) is a common pattern used with iOS and is integrated into the platform.

You are not limited to a specific pattern when building for UWP.

You can use the built-in data binding pattern to ensure clean separation of data concerns and UI concerns, and avoid having to code up UI event handlers which then update property values.

You can extend data binding to follow the **Model-View-ViewModel (MVVM)** pattern, either by making use of third-party MVVM libraries such as MVVM Light Toolkit, or rolling your own and keeping logic out of code-behind.

The MVVM Pattern

Template 10 Visual Studio project templates