AND210

Android Services

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft or Xamarin may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft or Xamarin, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

© 2014-2017 Xamarin Inc., Microsoft. All rights reserved.

Xamarin, MonoTouch, MonoDroid, Xamarin.iOS, Xamarin.Android, Xamarin Studio, and Visual Studio are either registered trademarks or trademarks of Microsoft in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

Objectives

- Create a service
- Start a service
- Elevate a service to foreground
- Bind to a service

Create a service

Xamarin University

Tasks

1. Define a service class
2. Override lifecycle methods
3. Declare a service in the app manifest

Motivation

- Applications may have long-running tasks that should not or cannot be interrupted

Encrypt sensitive data

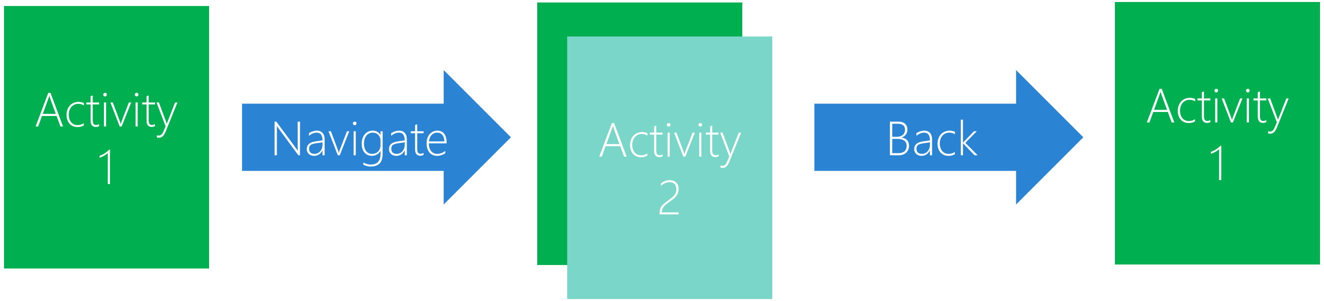
Monitor sensor input

Save user data or preferences

Play music

Activity lifespan

- Activities are created and destroyed based on the requirements of the application and user behavior

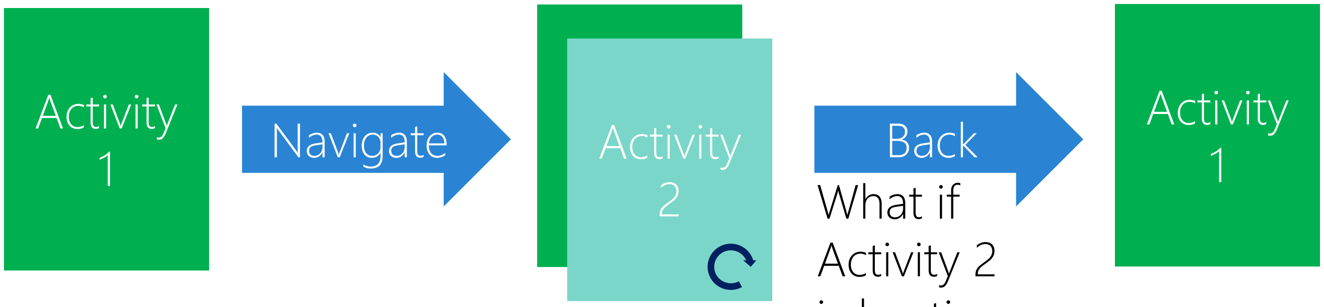


Activity 2 created

Activity 2 destroyed

Activity limitation

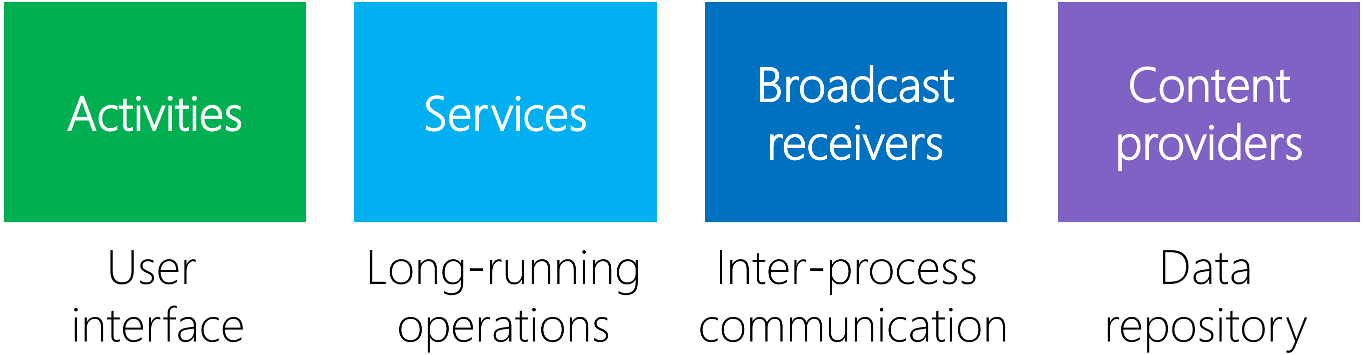
- Activities are inappropriate for hosting long-running tasks because they are destroyed as the user navigates through the back-stack



What if Activity 2 is hosting a long-running task?

Application components

- Android applications are made up of several *application components*



Activities

User interface

Services

Long-running operations

Broadcast receivers

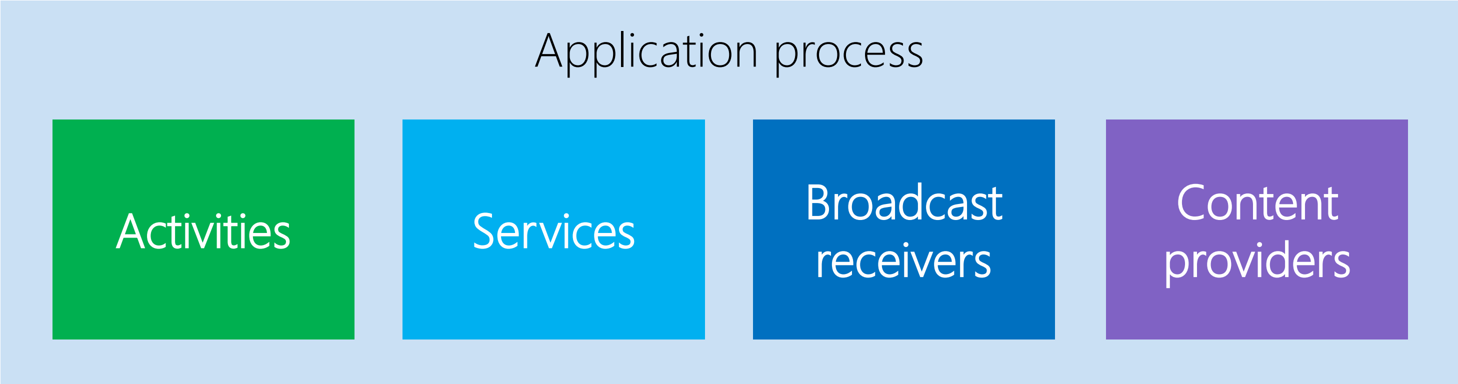
Inter-process communication

Content providers

Data repository

Application process

- Android applications run in their own process which executes the code in all of its application components



What is a service?

- A *service* is an Android application component that can perform long-running operations and does not require user interaction

Headless

Independent

Resilient

Headless

- A service does not present a UI directly – it can use feedback, notifications or running Activities to communicate with a user

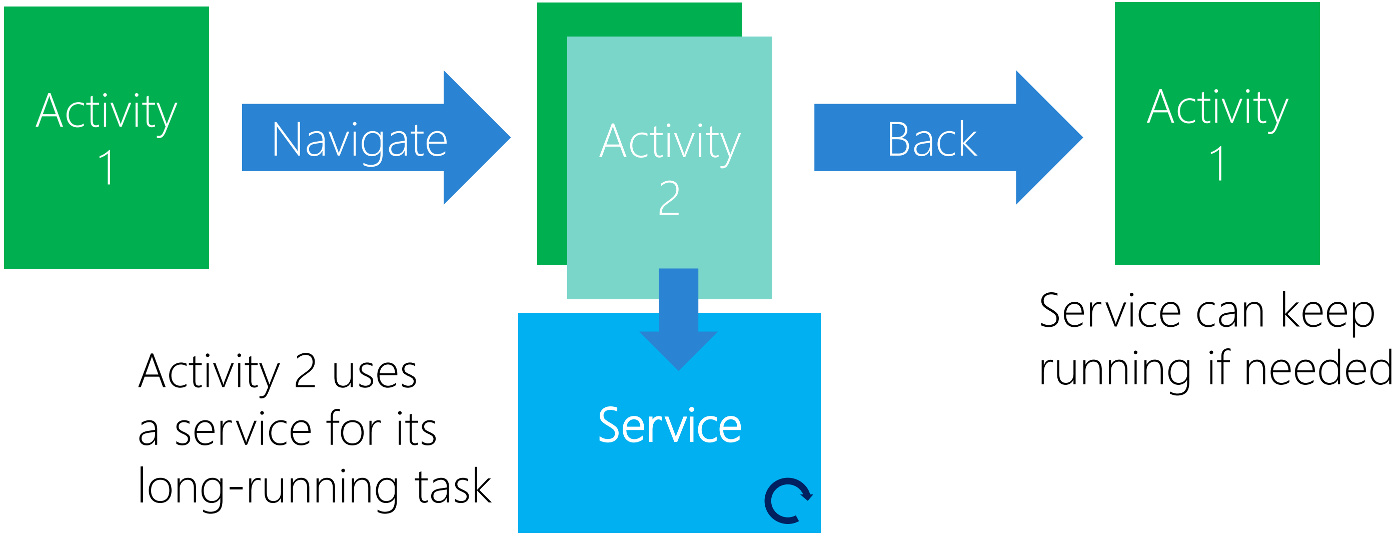
Display a toast

Display a Notification

Interact with an Activity

Independent

- Services can keep running even when the user switches apps or navigates to a different Activity

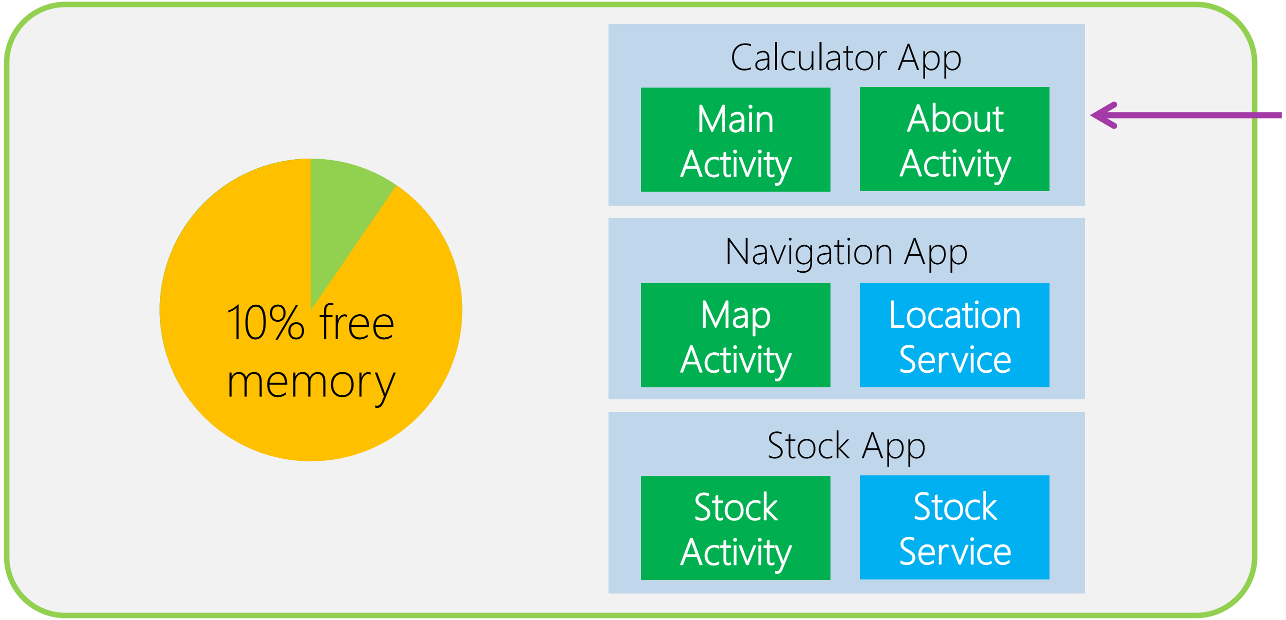


Activity 2 uses a service for its long-running task

Service can keep running if needed

Resilient

- Processes that host a running service have a higher priority and can be configured to automatically restart if killed



Apps without a service will be killed first

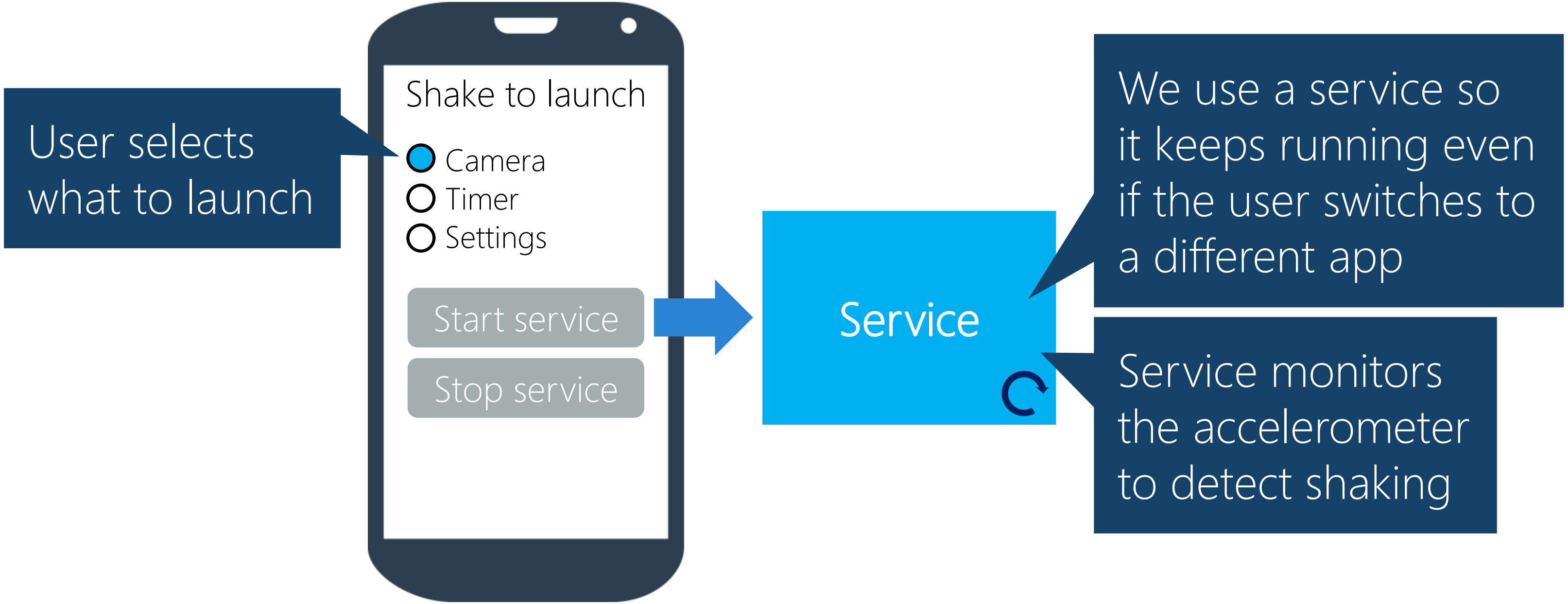
Demonstration

Running an application with a service

Xamarin University

Example service: shake-to-launch

- We will build a service that launches a function when the user shakes the device



User selects what to launch

Shake to launch

● Camera
○ Timer
○ Settings

We use a service so it keeps running even if the user switches to a different app

Service monitors the accelerometer to detect shaking

How to build a service

- Several steps are required to implement an Android service class

1. Derive from `Service`
2. Add the `Service` attribute
3. Code `OnCreate`
4. Code `OnStartCommand`
5. Code `OnDestroy`

1. Derive from Service [library class]

- All services derive from the abstract **`Service`** base class

```
public abstract class Service
{
   public virtual void OnCreate();
   public virtual void OnDestroy();

   public virtual StartCommandResult OnStartCommand(Intent intent, ...) {...}
   ...
}
```

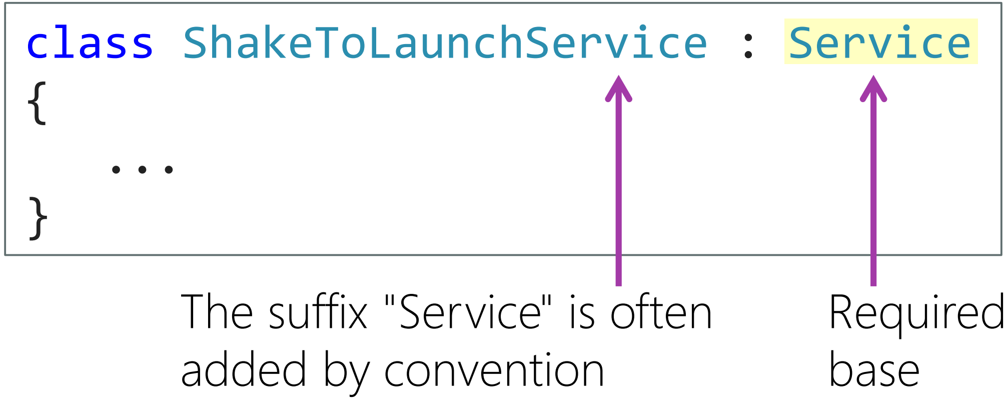
Defines overridable lifecycle callback methods

The return value controls the resiliency behavior

The Intent contains the "arguments", i.e. any input data the service needs to do its work

1. Derive from Service [your class]

- Your service must inherit from the library **Service** class



```
class ShakeToLaunchService : Service
{
    ...
}
```

The suffix "Service" is often added by convention

Required base

2. Add the Service attribute [motivation]

- A service must be declared in AndroidManifest.xml

```
<service android:name="string"
         android:description="string resource"
         android:label="string resource"
         android:icon="drawable resource"
         android:enabled=["true" | "false"] ... >
   ...
</service>
```

Service is identified by name

Attributes control service metadata and behavior

2. Add the Service attribute [use]

- Xamarin's **`Service`** attribute automatically adds the manifest entry

```
[Service Label="Shake to launch", Icon=Resource.Drawable.ic_vibration]
class ShakeToLaunchService : Service
{  ...
}
```

```
<manifest ...>
   <application ...>
      <service android:icon="@drawable/ic_vibration"
               android:label="Shake to launch"
               android:name="md5....ShakeToLaunchervice" />
   </application>
   ...
</manifest>
```

AndroidManifest is populated at build-time

3. Code OnCreate

- **OnCreate** performs any needed service initialization that is independent of the Intent used to start the service

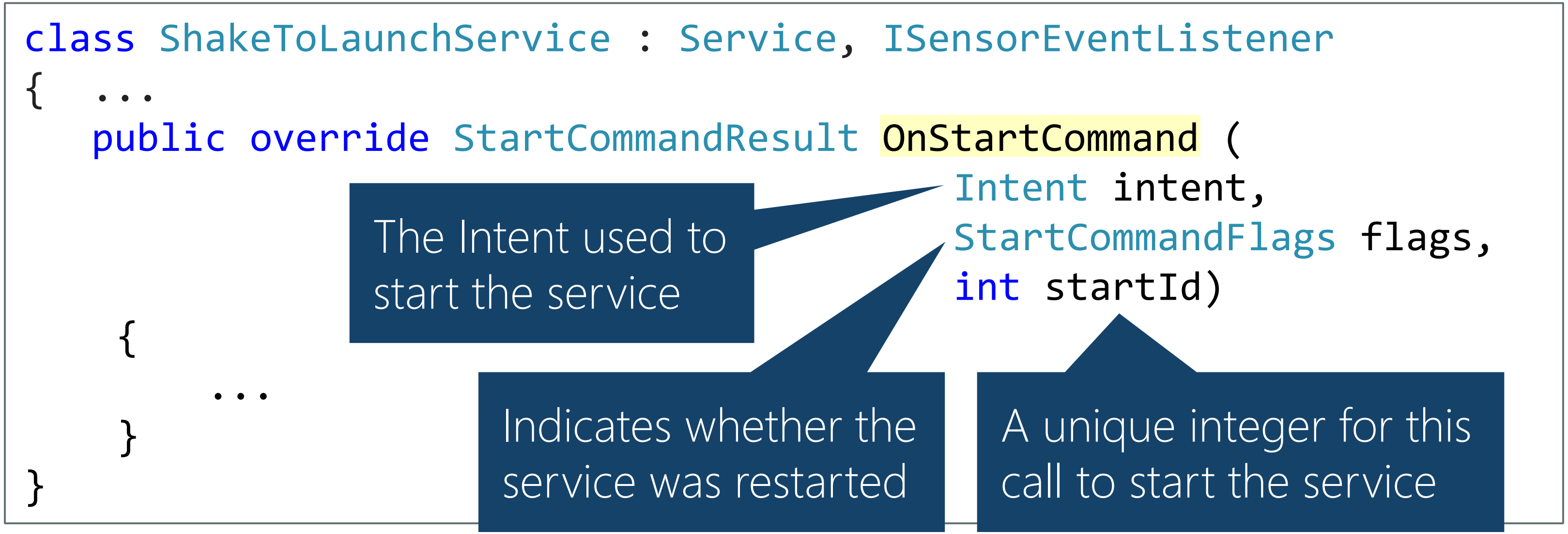
```
class ShakeToLaunchService : Service, ISensorEventListener
{  ...
   public override void OnCreate()
   {
      sensorManager = (SensorManager)GetSystemService(Context.SensorService);

      var accelerometer = sensorManager.GetDefaultSensor(SensorType.Accelerometer);

      if (accelerometer != null)
         sensorManager.RegisterListener(this, accelerometer, SensorDelay.Normal);
   }
}
```

This service uses the accelerometer to detect when the user shakes the device

The service class implements the sensor callback interface so it registers itself as a listener

4. Code OnStartCommand [signature]

- When a service is started, its **`OnStartCommand`** method is called which receives details about how the service was started



```
class ShakeToLaunchService : Service, ISensorEventListener
{  ...
   public override StartCommandResult OnStartCommand (
                                          Intent intent,
                                          StartCommandFlags flags,
                                          int startId)
    {
        ...
    }
}
```

The Intent used to start the service

Indicates whether the service was restarted

A unique integer for this call to start the service

4. Code OnStartCommand [return]

- **`OnStartCommand`** returns a **`StartCommandResult`** enumeration which determines how a started service behaves if it is stopped by the system

```
public override StartCommandResult OnStartCommand (...)
{
    ...
    return StartCommandResult
}
```

```
.Sticky
.NotSticky
.RedeliverIntent
.StickyCompatibility
```

Restarts with a blank intent

Won't restart

Restarts with the original intent

Xamarin University

4. Code OnStartCommand [work]

❖ **OnStartCommand** processes the input data from the Intent, then starts the actual work of the service

```
class ShakeToLaunchService : Service, ISensorEventListener
{  ...
   string intentAction;

   public override StartCommandResult OnStartCommand(Intent intent, ...)
   {
      intentAction = intent.GetStringExtra("Action");

      return StartCommandResult.RedeliverIntent;
   }

   public void OnSensorChanged(SensorEvent e) { ... }
}
```

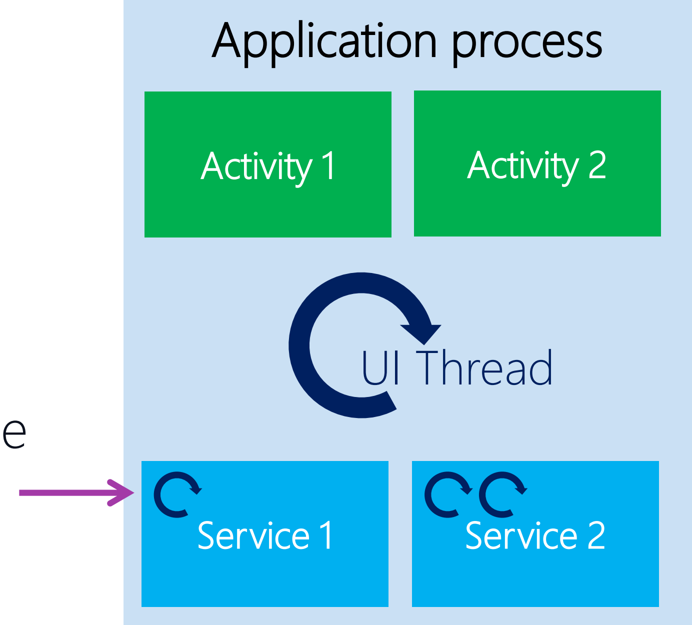
Client tells the service what to do when the user shakes the device e.g. start the camera

For this service, the remaining work is in the sensor callback

4. Code OnStartCommand [threading]

- A local service runs on the main thread of the hosting process – for applications this is the UI thread

Typical to create new threads to do your work



5. Code OnDestroy

- `OnDestroy` performs any needed cleanup – typically this involves stopping threads, unsubscribing, and/or releasing scarce resourced

```
class ShakeToLaunchService : Service, ISensorEventListener
{  ...
   public override void OnDestroy()
   {
      sensorManager.UnregisterListener(this);

      base.OnDestroy();
   }
}
```

Unsubscribe from sensor events

Exercise

Create a service

Summary

1. Define a service class
2. Override lifecycle methods
3. Declare a service in the app manifest

Start a service

Tasks

1. Start a service
2. Stop a service

Motivation

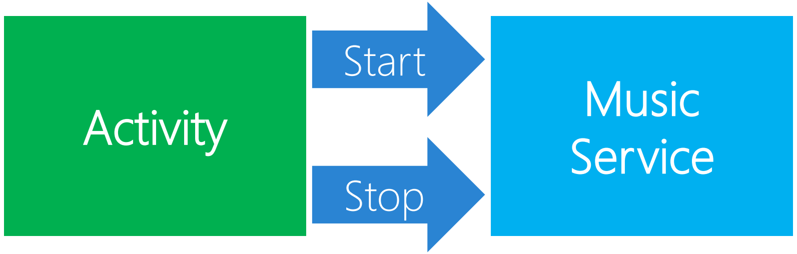
- Services can be used to perform long-running operations independent of your application UI – even when your application is backgrounded

Download

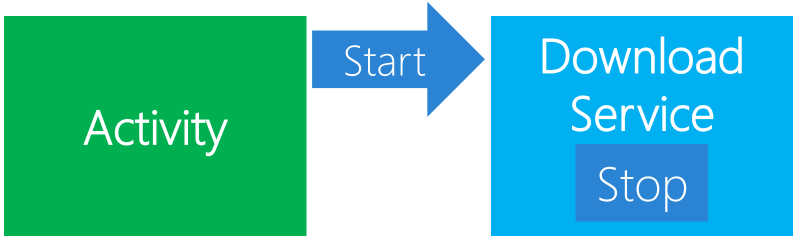
Play
music

What is a started service?

- A *started service* is a service that runs independently of other application components



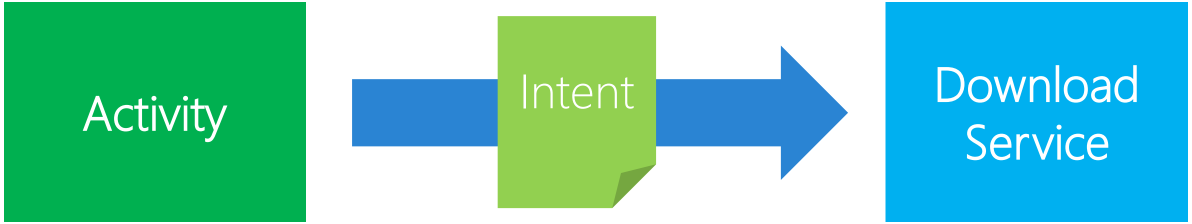
Can be started and stopped explicitly



Can be started explicitly and then stop itself

Service input data

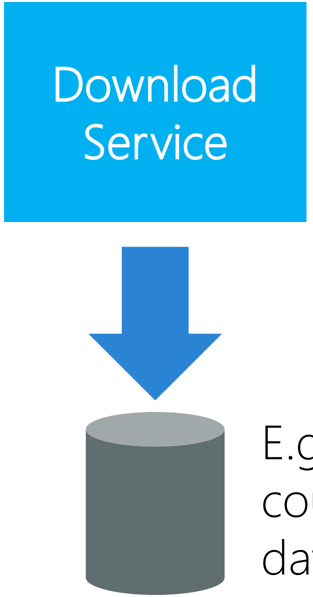
- Clients use an Intent to access a service – the Intent can contain input data (i.e. the arguments) that tell the service what to do



Intent Extras could contain the URL for the file to download

Started service output [file system]

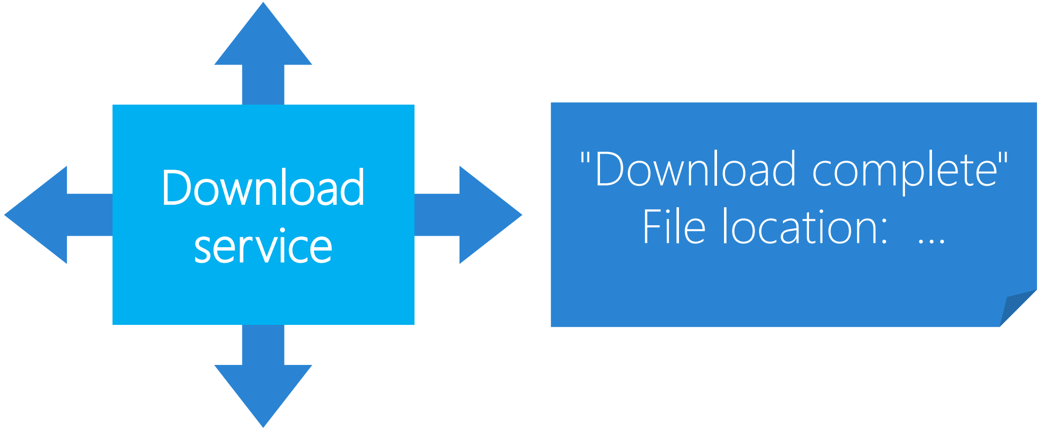
- Started services can write their results to the file system



E.g. Download Service could write the downloaded data to a file

Started service output [broadcast]

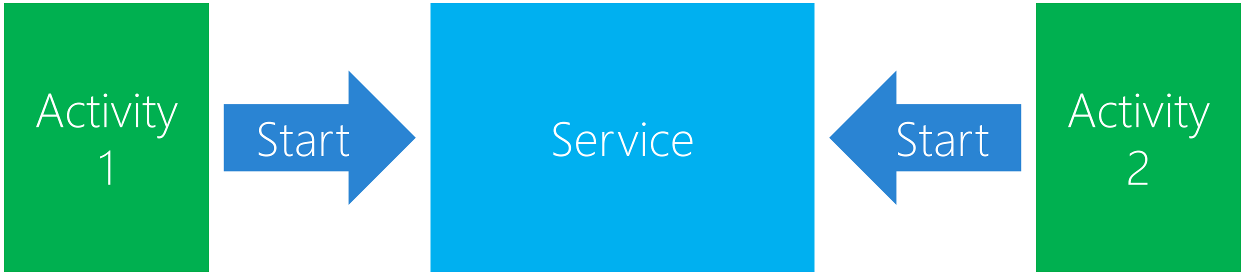
- Started services can broadcast a message to other application components to notify them of progress or pass results



See: https://developer.xamarin.com/api/type/Android.Content.BroadcastReceiver

Single instance

- At most once instance of each service type can run



Activity 2 will use the same service instance

Service control methods

❖ The methods to control a service come from **Context** and **Service**

External control

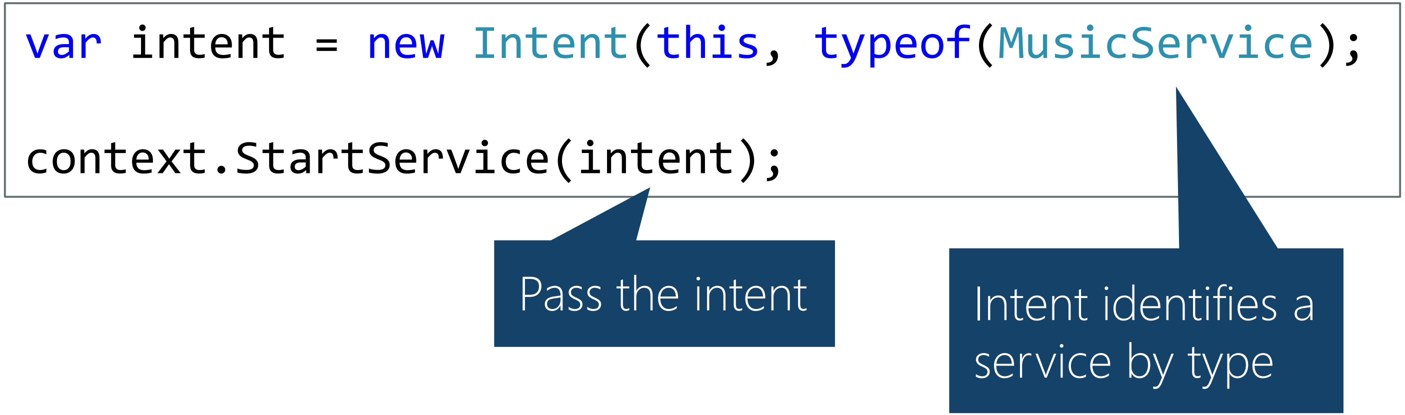
```
public abstract class Context
{  ...
   public abstract ComponentName StartService(Intent service);
   public abstract bool          StopService (Intent service);
}
```

Service can stop itself

```
public abstract class Service
{  ...
   public void StopSelf();
}
```

Start a service

❖ To start a service, call **StartService** and pass an **Intent** to identify the service



```
var intent = new Intent(this, typeof(MusicService));

context.StartService(intent);
```

Pass the intent

Intent identifies a service by type

Stop a service [external]

- A started service can be stopped from a context by calling the `StopService` method

```
var intent = new Intent(this, typeof(MusicService));

context.StopService(intent);
```

Does not need to be the same intent instance that started the service

The Intent identifies the service

Stop a service [internal]

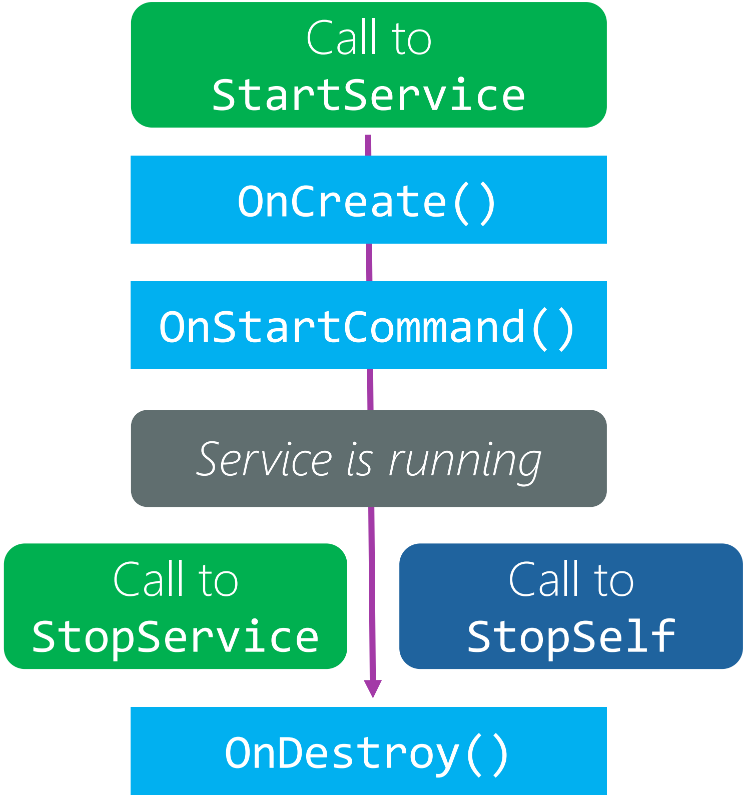
- A service can stop itself by calling its **`StopSelf`** method

```
class DownloadService : Service
{  ...
   void DownloadCompleted ()
   {  ...
      this.StopSelf();
   }
}
```

The service determines when its task is done

Started service lifecycle

- A started service follows a well defined lifecycle - independent of the application component that started the service



Exercise

Start a service

Xamarin University

Summary

1. Start a service
2. Stop a service

Elevate a service to foreground

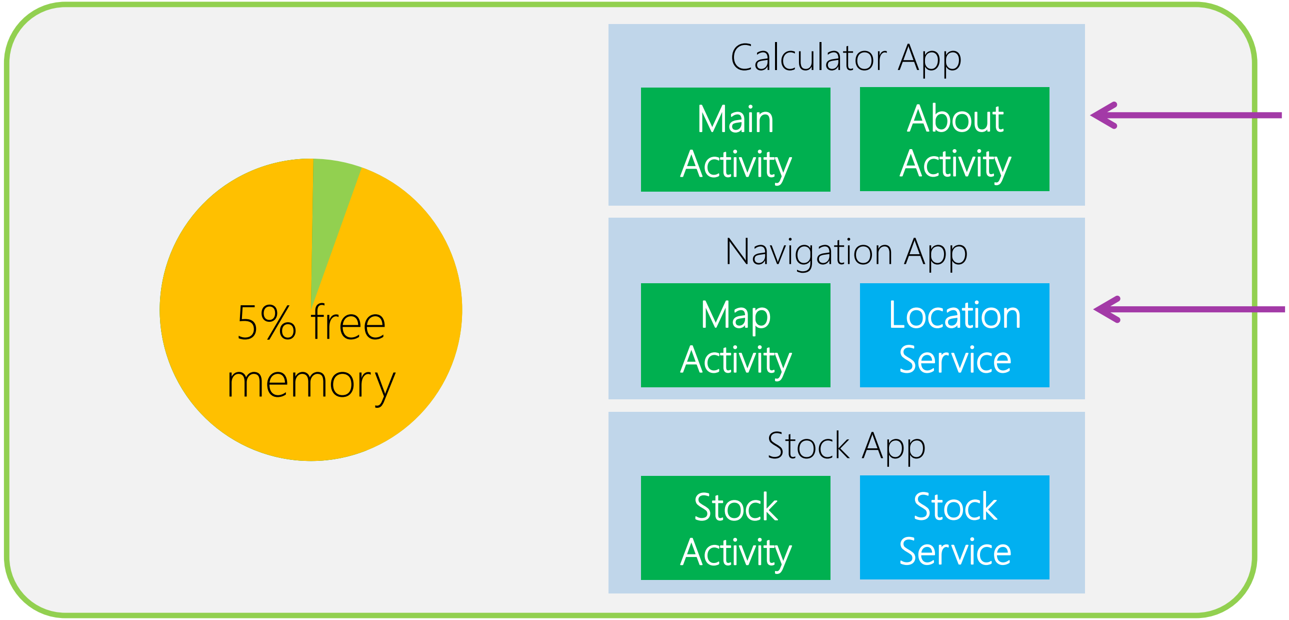
Xamarin University

Tasks

1. Bring a service to the foreground
2. Build a notification
3. Update a notification
4. Launch an Activity from a notification
5. Remove a service from the foreground

Motivation

- Android might kill app components containing services if it gets low on resources



Apps without a service will be killed first

Even apps with services may be killed

User-aware services

- Some services perform work that is visible to the user – we would prefer Android not kill these services when low on resources

Download apps or application content

Make VOIP calls

Play or stream music

What is a foreground service?

- A *foreground service* is a service that runs at a higher priority and displays a notification on the status bar

```
public abstract class Service
{  ...
   public void StartForeground (int id, Notification notification);
   public void StopForeground (bool removeNotification);
}
```

Generally, a service elevates itself to the foreground

A notification is required so the user is aware that the service is now foreground

What is a notification?

- A `Notification` contains a message that can be displayed to the user outside of your application's UI

Sound or vibration

Flashing LED or backlight

Status bar icon

Creating a notification

- Create a notification by instantiating a **`Notification.Builder`** object and calling its **`Build`** method

```
Notification myNotification = new Notification.Builder(this)
                                 .SetContentTitle(tag)
                                 .SetContentText(content)
                                 .SetSmallIcon(Resource.Drawable.NotifySm)
                                 .Build();
```

Builder uses a fluent API to set properties and assign visual resources

Call **`Build`** to create the **`Notification`**

How to enter foreground

- A service becomes a foreground service by calling its **`StartForeground`** method and passing in a **`Notification`**

```
public class MyService : Service
{
   void Enter ()
   {
      var myNotification = new Notification.Builder(this)...Build();

      this.StartForeground(NotificationID, myNotification);
   }
}
```

Arbitrary unique (non-zero) integer

Updating a notification

- The **NotificationManager** can update an active notification by sending a new notification with the original unique ID

```
var updatedNotification = new Notification.Builder....Build();

var manager = (NotificationManager)GetSystemService(Context.NotificationService);

manager.Notify(NotificationID, updatedNotification);
```

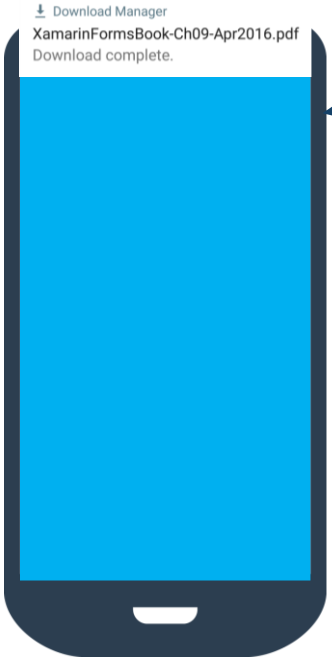
Create a new **Notification**

Retrieve the notification manager service

Call **Notify** with the same ID used with the original notification

Responding to Notifications

- It is common for a notification to respond to taps by launching a relevant Activity



Download Manager

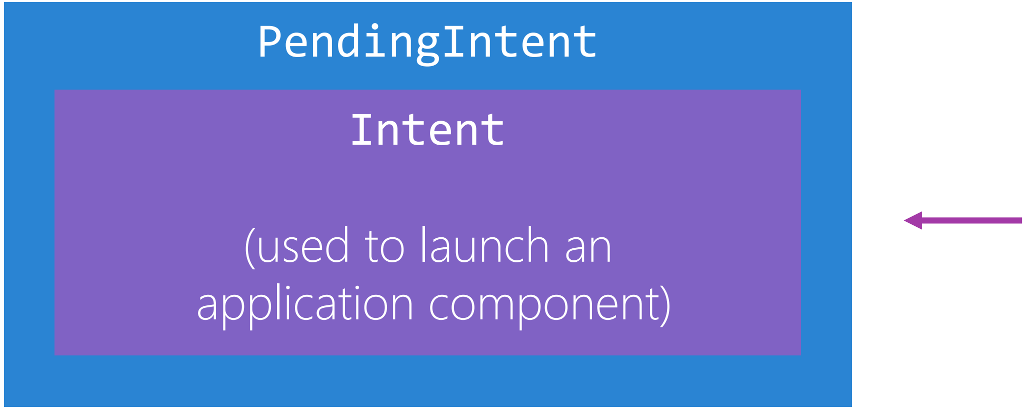
XamarinFormsBook-Ch09-Apr2016.pdf

Download complete.

User expects that tapping on the notification will let them access the file

What is a PendingIntent?

- A **`PendingIntent`** allows your application to give an external process permission to perform an operation on its behalf - using the application’s identity and permissions



`PendingIntent`s wrap an **`Intent`** which defines the operation to perform

Create a PendingIntent

- **PendingIntents** are created by calling one of four factory methods on the **PendingIntent** class

Intent to launch an **Activity**

```
var intent = new Intent(this, typeof(DisplayFileActivity));

var pendingIntent = PendingIntent.GetActivity(this, intent, 0);
```

Intent is passed into factory method

PendingIntent also includes factory methods to start multiple activities, perform a broadcast or start a **Service**

Include an Intent in a notification

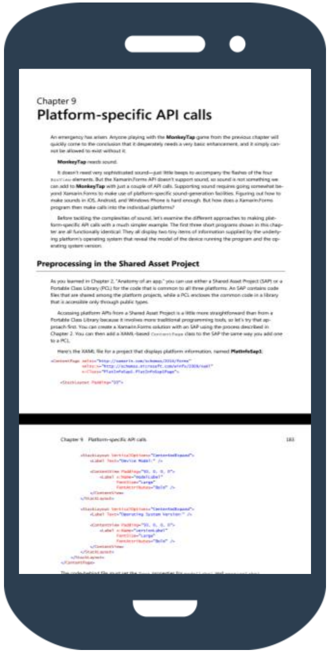
- A **PendingIntent** can be assigned to a **Notification** to allow the notification to do work on behalf of your application

```
var pendingIntent = PendingIntent.GetActivity(this, intent, 0);

var Notification.Builder(this)
                   .SetContentTitle("My Service")
                   .SetSmallIcon(Resource.Drawable.NotifyIcon)
                   .SetContentIntent(pendingIntent).Build();
```

PendingIntent allows the Notification to launch an **Activity**

Launching an Activity from a notification

- The Activity in the notification's **`PendingIntent`** is launched automatically when the user taps the notification



Chapter 9

Platform-specific API calls

An emergency has arisen. Anyone playing with the **MonkeyTap** game from the previous chapter will quickly come to the conclusion that it desperately needs a very basic enhancement, and it simply cannot be allowed to exist without it.

MonkeyTap needs sound.

It doesn't need very sophisticated sound—just little beeps to accompany the flashes of the four `BoxView` elements. But the Xamarin.Forms API doesn't support sound, so sound is not something we can add to **MonkeyTap** with just a couple of API calls. Supporting sound requires going somewhat beyond Xamarin.Forms to make use of platform-specific sound-generation facilities. Figuring out how to make sounds in iOS, Android, and Windows Phone is hard enough. But how does a Xamarin.Forms program then make calls into the individual platforms?

Before tackling the complexities of sound, let's examine the different approaches to making platform-specific API calls with a much simpler example. The first three short programs shown in this chapter are all functionally identical. They all display two tiny items of information supplied by the underlying platform's operating system that reveal the model of the device running the program and the operating system version.

Preprocessing in the Shared Asset Project

As you learned in Chapter 2, "Anatomy of an app," you can use either a Shared Asset Project (SAP) or a Portable Class Library (PCL) for the code that is common to all three platforms. An SAP contains code files that are shared among the platform projects, while a PCL encloses the common code in a library that is accessible only through public types.

Accessing platform APIs from a Shared Asset Project is a little more straightforward than from a Portable Class Library because it involves more traditional programming tools, so let's try that approach first. You can create a Xamarin.Forms solution with an SAP using the process described in Chapter 2. You can then add a XAML-based `ContentPage` class to the SAP the same way you add one to a PCL.

Here's the XAML file for a project that displays platform information, named **PlatInfoSap1**:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="PlatInfoSap1.PlatInfoSap1Page">

    <StackLayout Padding="20">
```

Chapter 9 Platform-specific API calls 183

```
        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="Device Model:" />

            <ContentView Padding="50, 0, 0, 0">
                <Label x:Name="modelLabel"
                       FontSize="Large"
                       FontAttributes="Bold" />
            </ContentView>
        </StackLayout>

        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="Operating System Version:" />

            <ContentView Padding="50, 0, 0, 0">
                <Label x:Name="versionLabel"
                       FontSize="Large"
                       FontAttributes="Bold" />
            </ContentView>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

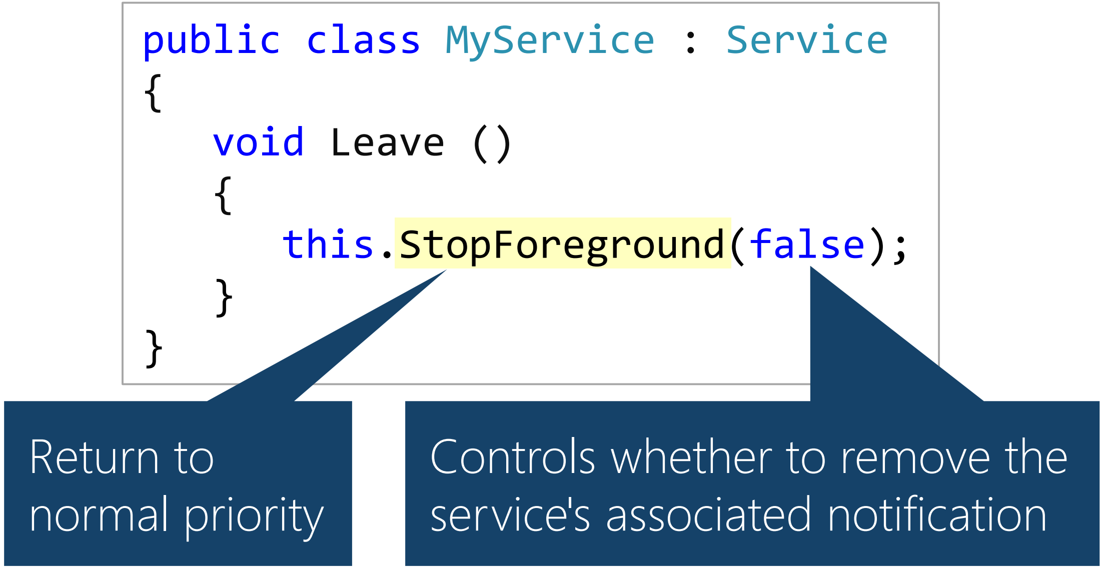
The code-behind file must set the `Text` properties for `modelLabel` and `versionLabel`

User taps the notification…

…and the Activity starts

How to leave foreground

- A service leaves the foreground by calling its **`StopForeground`** method – this does not stop the service



```
public class MyService : Service
{
    void Leave ()
    {
        this.StopForeground(false);
    }
}
```

Return to normal priority

Controls whether to remove the service's associated notification

Exercise

Elevate a service to foreground

Summary

1. Bring a service to the foreground
2. Build a notification
3. Update a notification
4. Launch an Activity from a notification
5. Remove a service from the foreground

Bind to a service

Xamarin University

Tasks

1. Create a service binder
2. Create a service connection
3. Bind to service
4. Respond to binding notifications
5. Introduce hybrid services

Motivation

- Activities may need to interact with a running service

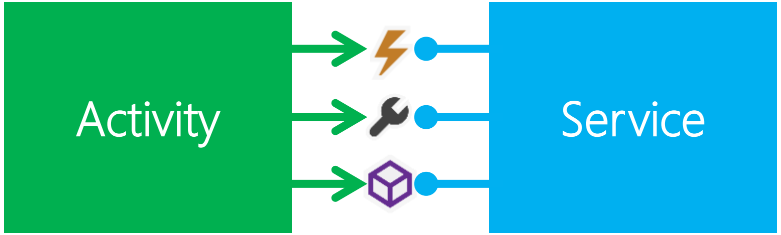
Synchronize application data

Display sensor data

Process local data

What is a bound service? [access]

- A *bound* service is a service that is reachable by the client

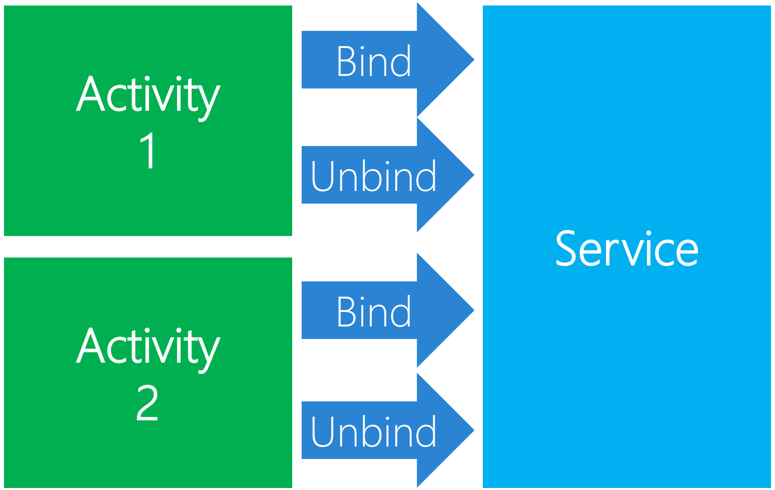


Activity can use the service's events, properties, and methods

Xamarin University

What is a bound service? [lifetime]

- A bound service lives only as long as there are bound clients



Service created on first bind
destroyed when last client unbinds

Xamarin University

Inter-process communication (IPC)

- Binding underlies all IPC in Android so the APIs are relatively complex

```
var manager = (SensorManager)GetSystemService(Context.SensorService);
```

System services run in their own process – binding is used for the IPC

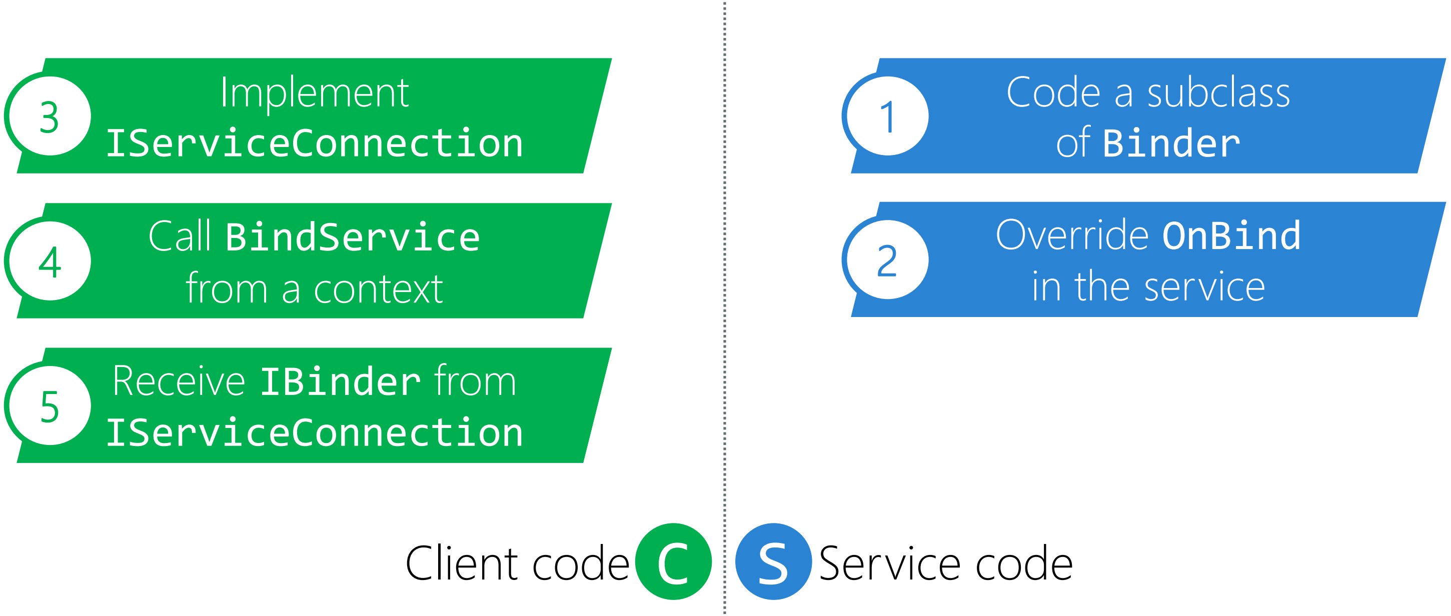
```
var intent = new Intent(Intent.ActionView);
intent.SetData(Android.Net.Uri.Parse("http://www.microsoft.com"));
StartActivity(intent);
```

Intents are implemented using binding

This course uses private services that run in the same process as the client app

Binding to a service

- Enabling a bound service requires both the service and client to write code



3 Implement **`IServiceConnection`**

4 Call **`BindService`** from a context

5 Receive **`IBinder`** from **`IServiceConnection`**

Client code C

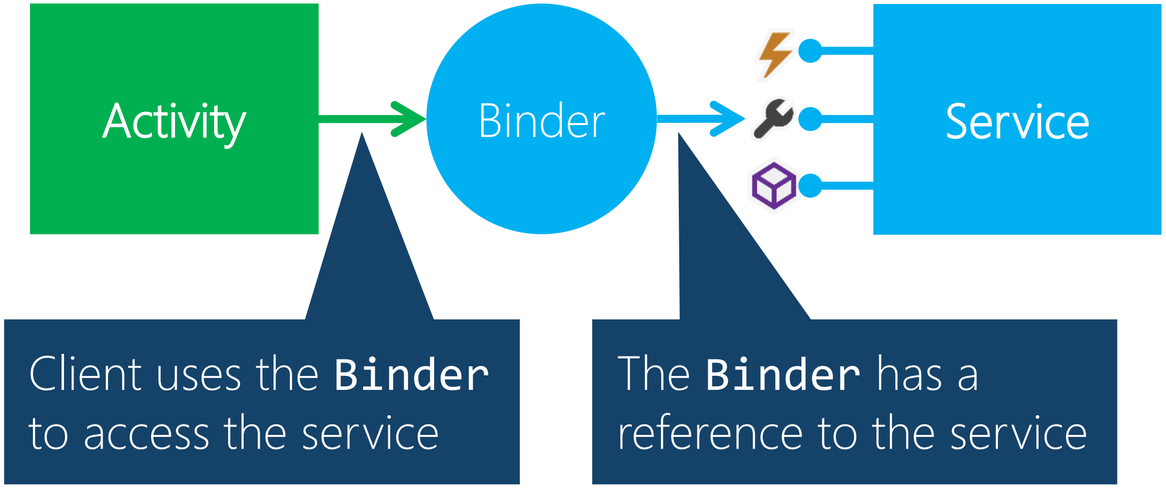
1 Code a subclass of **`Binder`**

2 Override **`OnBind`** in the service

S Service code

Client access

- The service must provide an **`IBinder`** object that gives the client access to the service



Client uses the **`Binder`** to access the service

The **`Binder`** has a reference to the service

Xamarin University

What is IBinder?

- **IBinder** defines the interface for Android remote-procedure call (RPC)

```
public interface IBinder : IJavaObject, IDisposable
{
   string InterfaceDescriptor { get; }
   IInterface QueryLocalInterface(string descriptor);
   void Dump     (FileDescriptor fd, string[] args);
   void DumpAsync(FileDescriptor fd, string[] args);
   bool IsBinderAlive { get; }
   void LinkToDeath  (IBinderDeathRecipient recipient, int flags);
   bool UnlinkToDeath(IBinderDeathRecipient recipient, int flags);
   bool PingBinder();
   bool Transact(int code, Parcel data, Parcel reply, TransactionFlags flags);
}
```

IBinder is complex since it handles IPC – e.g. **Transact** is a remote procedure call where the **Parcel**s are the arguments and return data

What is Binder?

- The library **Binder** class is the standard implementation of **IBinder** that handles the complexity of IPC for you

```
public class Binder : IBinder
{
   ...
}
```

The implementation is difficult – Android guidance is that you should use this class and not try to implement **IBinder** yourself

Code a subclass of Binder

- You write a **Binder** subclass that wraps an instance of the service

```
public class StepServiceBinder : Binder
{
   public StepCounterService Service { get; private set; }

   public StepServiceBinder(StepCounterService service)
   {
      this.Service = service;
   }
}
```

Inherit from **Binder**

Property exposes the service

Service is set in the constructor

C S

Override OnBind

- A service's **OnBind** method is called when the service is bound – it returns an **IBinder** to provide the client access to the service

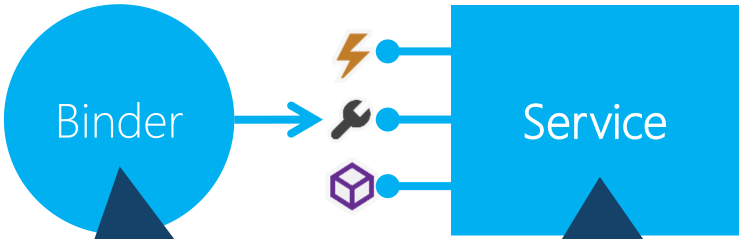
```
public class StepCounterService : Service
{
    public override IBinder OnBind (Intent intent)
    {
        return new StepServiceBinder(this);
    }
    ...
}
```

You create the binder...

...and pass it the service instance

Binder review

- The service creates a **Binder** subclass instance that contains a reference to the service

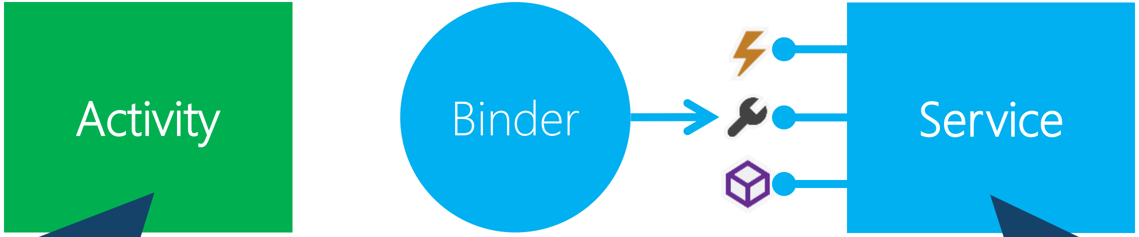


The binder constructor stores the service reference for later use by the client

The service's **OnBind** method creates the binder and passes itself to the constructor

Binding is asynchronous

- Binding creation is asynchronous because it needs to work cross-process and creating the target service can take time

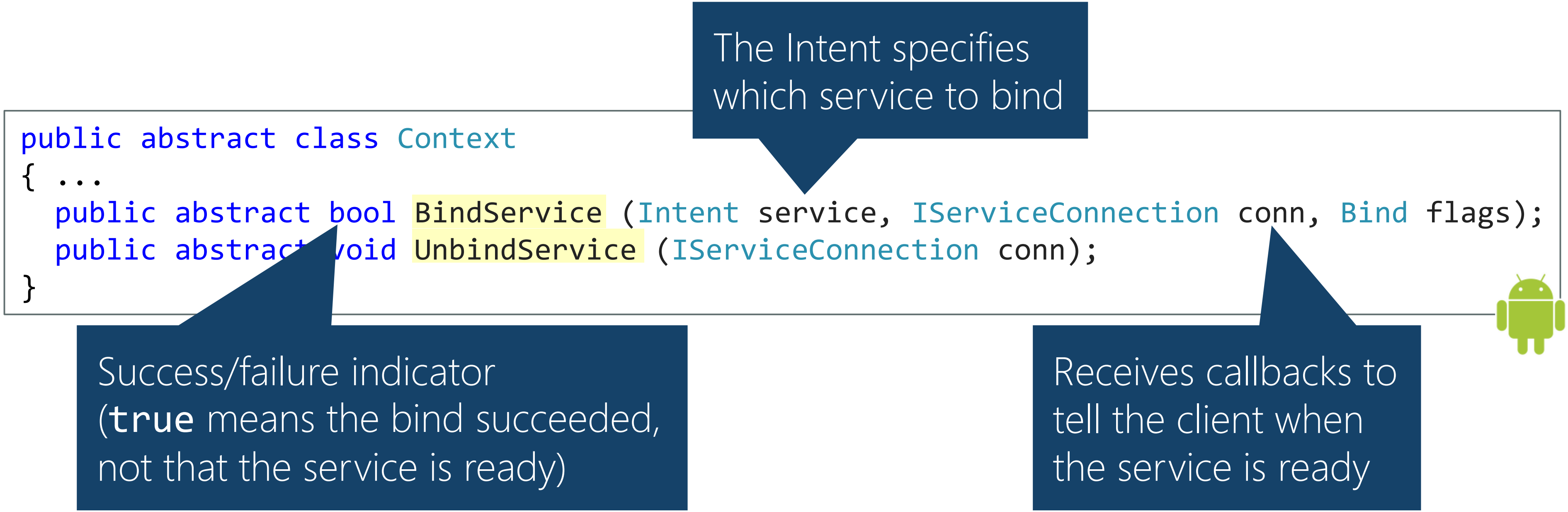


Client attempts to bind a service, the call returns immediately and there is no service yet

Once the service and binder are ready, the client receives a callback

Client binding methods

- The client binds/unbinds using methods from **`Context`**



```
public abstract class Context
{ ...
  public abstract bool BindService (Intent service, IServiceConnection conn, Bind flags);
  public abstract void UnbindService (IServiceConnection conn);
}
```

The Intent specifies which service to bind

Success/failure indicator (**`true`** means the bind succeeded, not that the service is ready)

Receives callbacks to tell the client when the service is ready

What is IServiceConnection?

- `IServiceConnection` is an interface for monitoring the state of a bound service – the client implements it to be notified when a bound service is ready

```
public interface IServiceConnection : IJavaObject, IDisposable
{
   void OnServiceConnected(ComponentName name, IBinder service);
   void OnServiceDisconnected(ComponentName name);
}
```

The methods are called when a service is bound or unbound

The binder object contains a reference to the service

Implement IServiceConnection

- The client implements **`IServiceConnection`** so it knows when the service is ready and can then get access to the service

```
class StepServiceConnection : Java.Lang.Object, IServiceConnection
{ ...
   public void OnServiceConnected(ComponentName name, IBinder service)
   {
      var stepBinder = service as StepServiceBinder;

      StepService stepService = stepBinder.Service;
      ...
   }
}
```

Contains the service but is *not* the service

Service is exposed via the public **`Service`** property

Cast to specific **`IBinder`** implementation

Bind a service

- A service is bound when an application component calls the **`BindService`** method

`Intent` specifies the service type

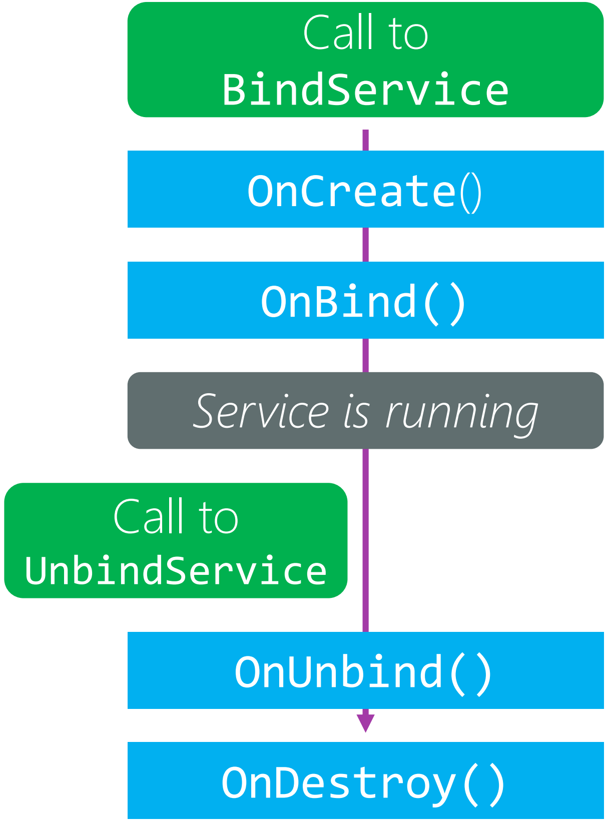
```
var intent = new Intent(this, typeof(StepCounterService));

var serviceConnection = new StepServiceConnection();

context.BindService(intent, serviceConnection, Bind.AutoCreate);
```

The **`IServiceConnection`** implementation for notification when the service is ready

Use **`Bind.AutoCreate`** for local services

Bound service Lifecycle

- A bound service follows a well-defined lifecycle

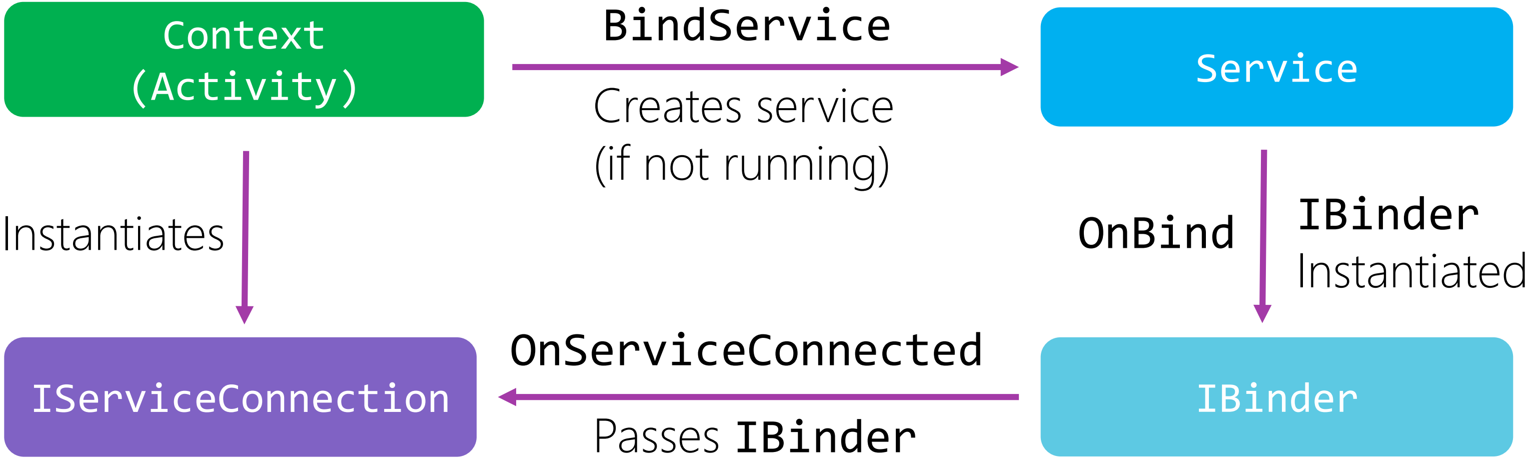


Exercise

Bind to a service

Bound service big picture

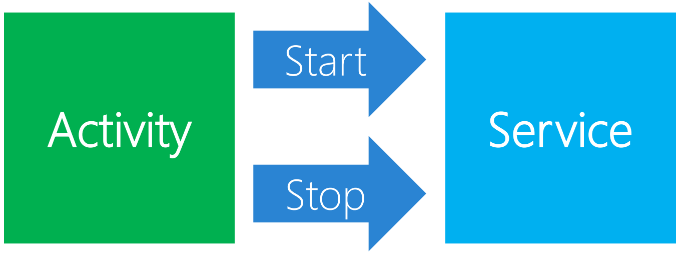
- Binding to a service involves four players: the **application component** binding to the service, the **service** itself, the **binder** and the **service connection**



Started vs bound service

- Started services and bound services differ in lifetime and client access

Started



Activity

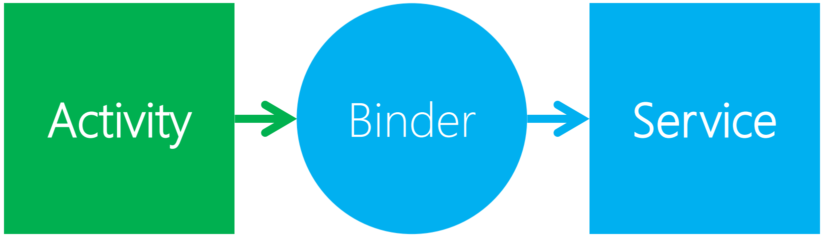
Start

Stop

Service

Lifetime: started and stopped explicitly
Access: no client access to service

Bound



Activity

Binder

Service

Lifetime: auto-stop when last client unbinds
Access: client accesses service through binder

Hybrid services

- It is possible to both start and bind to a service – this is a *hybrid service*

```
var intent = new Intent(this, typeof(StepCounterService));

var serviceConnection = new StepServiceConnection();

context.StartService(intent);
...
context.BindService(intent, serviceConnection, Bind.AutoCreate);
```

A hybrid service can be started first and then bound only when interaction or visualization is required

A hybrid service will only be destroyed if the service is both stopped AND all subscribers have unbound

Demonstration

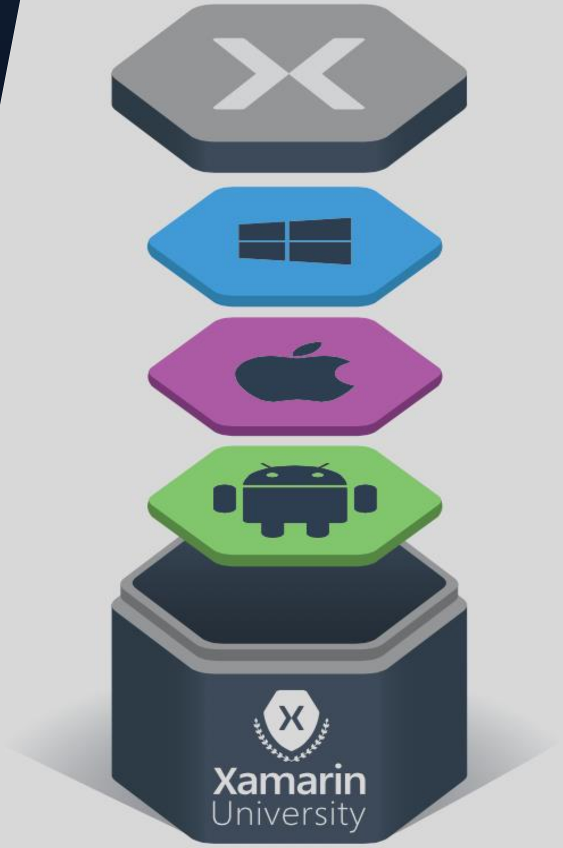
Creating a hybrid service

Summary

1. Create a service binder
2. Create a service connection
3. Bind to service
4. Respond to binding notifications
5. Introduce hybrid services

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile



Microsoft