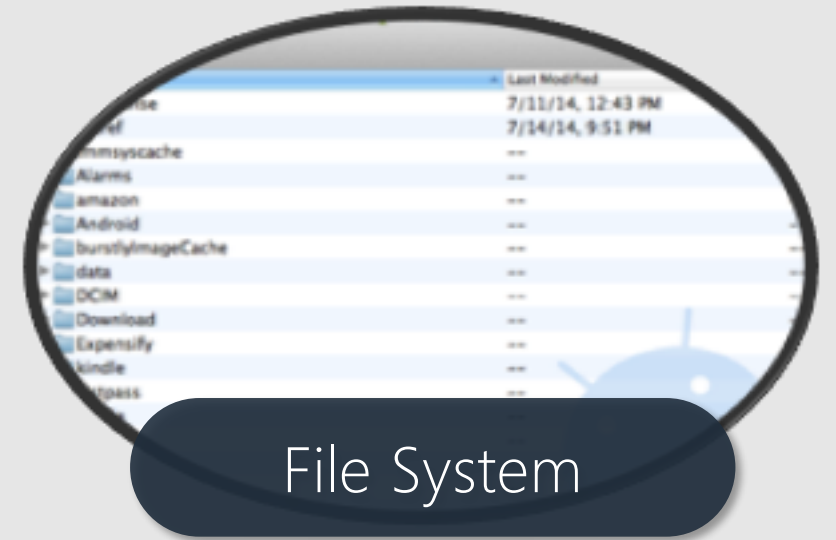
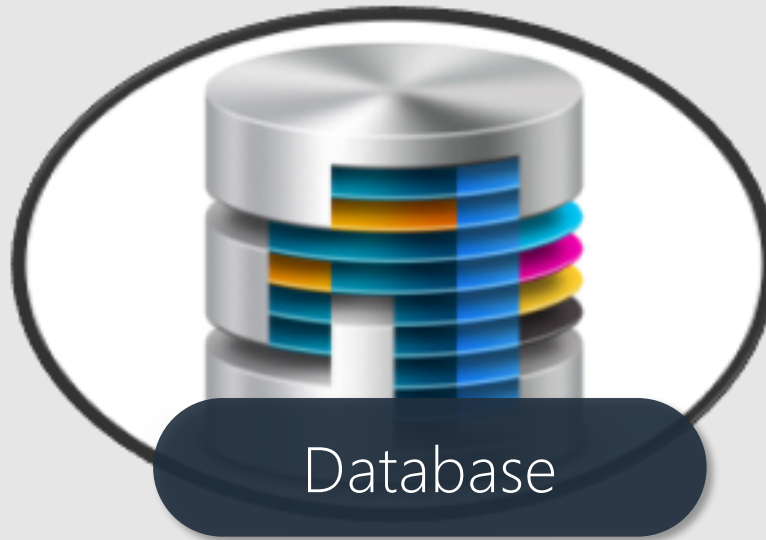
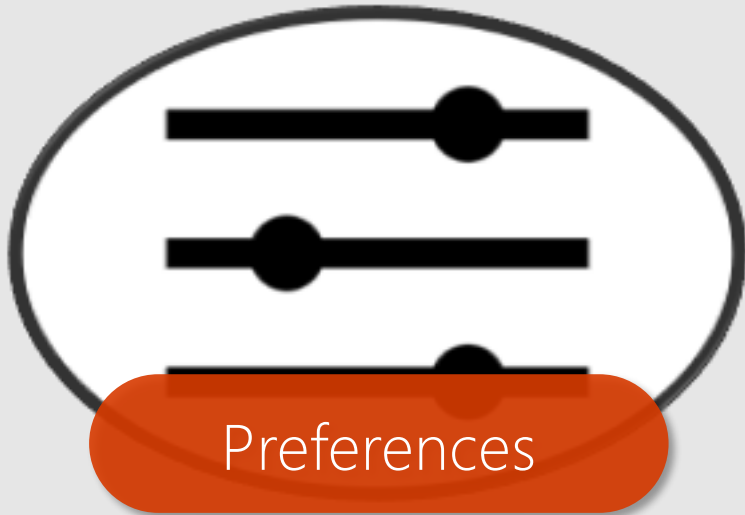


Working with Local Data

Data-storage options

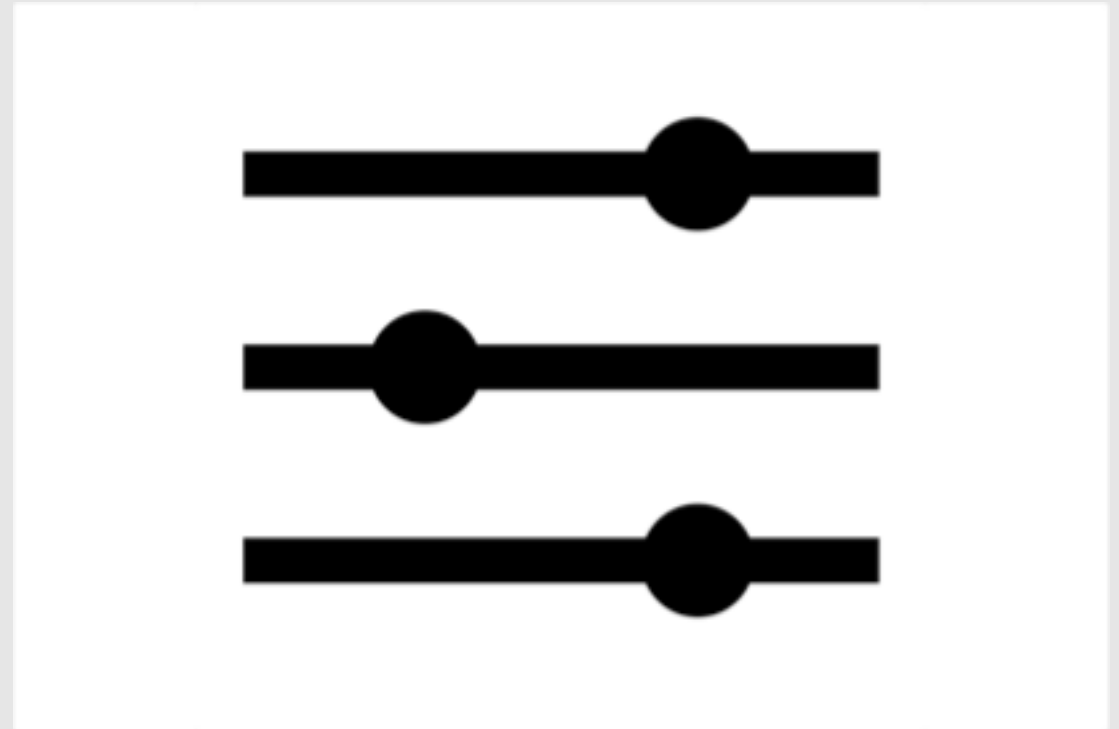
You have several options to store local information



Preferences

iOS, Android, and UWP provide native APIs to store **app-specific settings** as key-value pairs

- ✓ app configuration
- ✓ user preferences



Cross-platform settings support

Nuget package `Xam.Plugins.Settings` enables platform-agnostic storage

```
public static class MySettings
{
    const string NameKey = "userName";

    public static string Name
    {
        get { return CrossSettings.Current.GetValueOrDefault<string>(NameKey, ""); }
        set { CrossSettings.Current.AddOrUpdateValue <string>(NameKey, value); }
    }
}
```

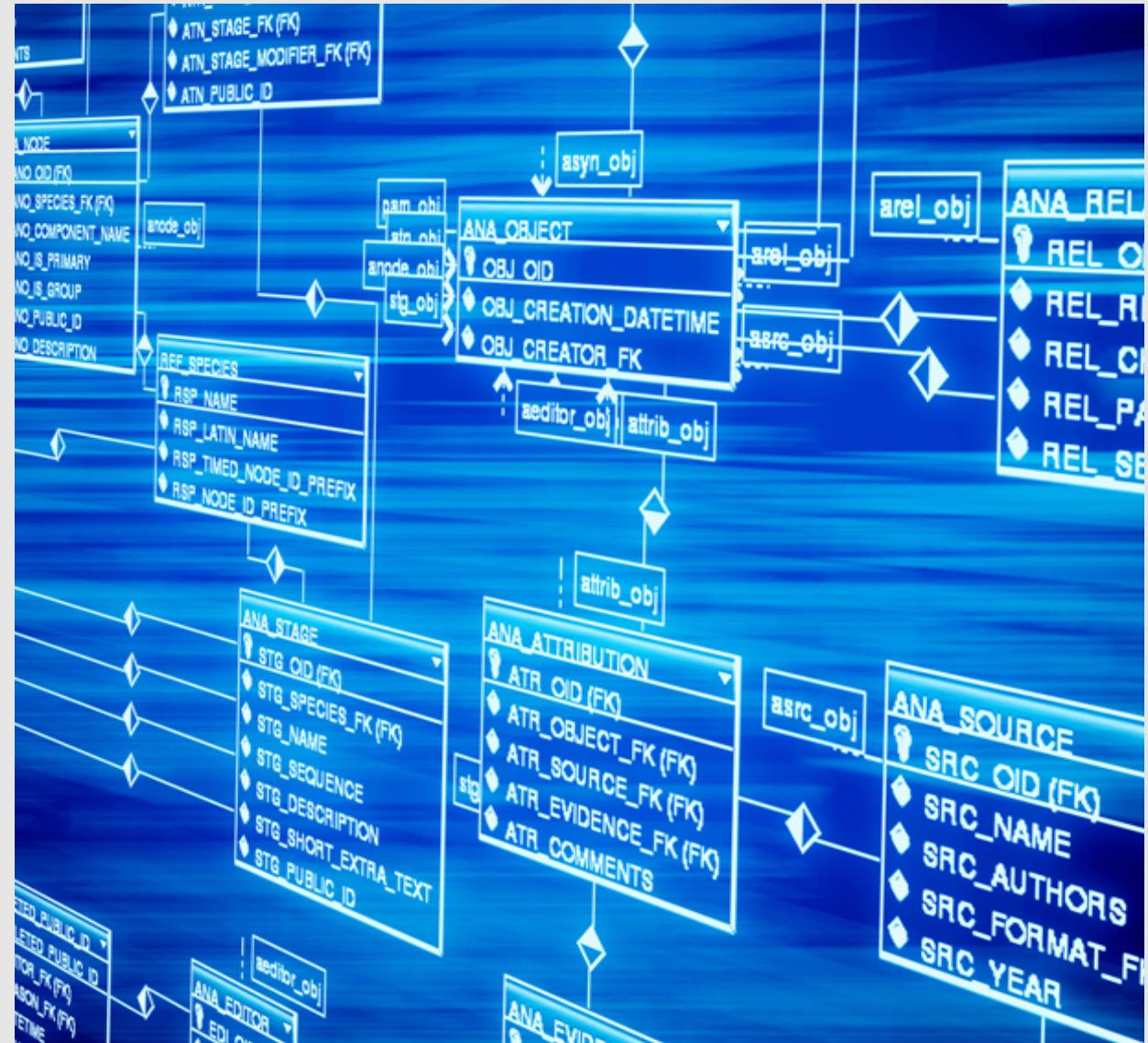
Plugin provides simple
get/set API that
does the persistent
storage for you

Using a database

iOS and Android include **SQLite**; an optional Nuget package provides support for Windows and WinPhone

Or can use NoSQL approach with **Couchbase**

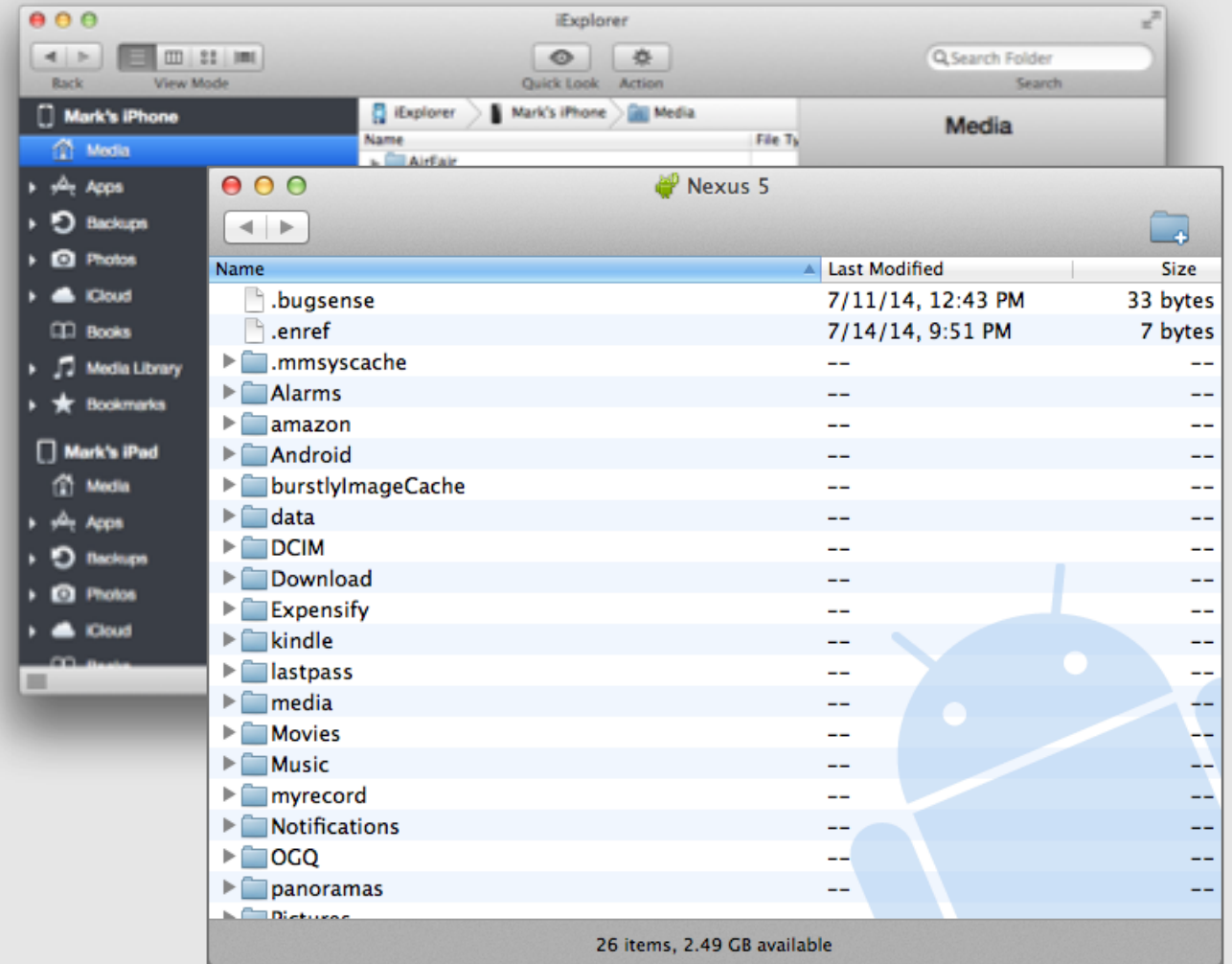
Can also store in the cloud – Azure Mobile Services, Amazon, Dropbox, etc.



The file system

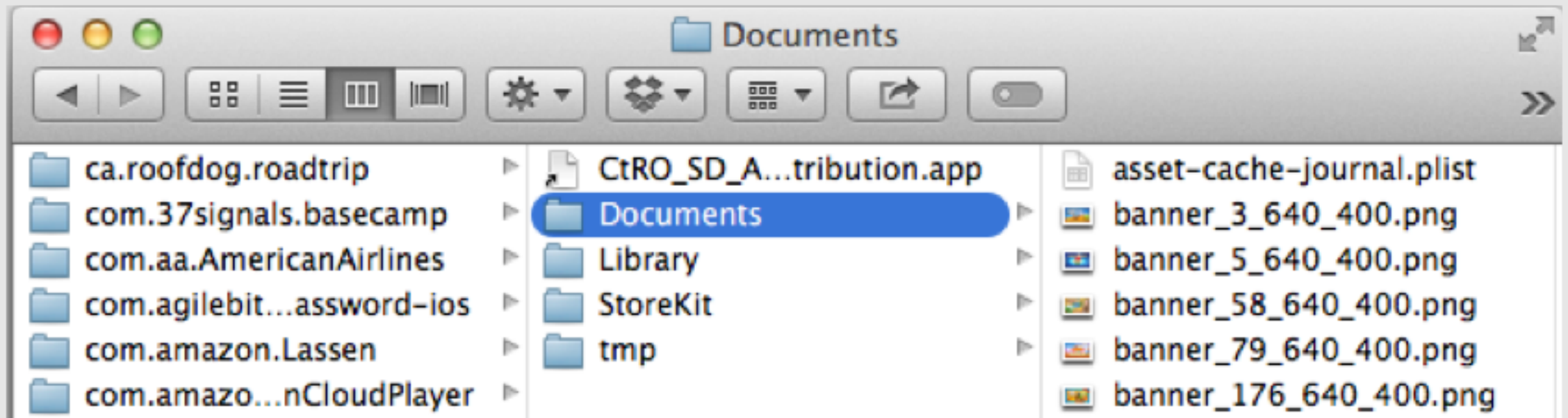
Devices have persistent file systems to store settings, applications, data, etc.

File system structure and content vary based on the operating system



The app sandbox

Your application is given a dedicated folder, called the **app folder** or **sandbox**, on the file system which contains **app-specific content**



Each iOS application has a folder, which contains sub-folders, which, in turn, contain your data and assets you create

Working with Files and Folders

Can work directly with files and folders using classes in **System.IO** namespace

```
using System.IO;
...
public IEnumerable<string> LoadSurvey(string filename)
{
    StreamReader reader = File.OpenText(filename);
    ...
}
```


Working with native APIs

Can use platform-specific APIs to access unique features, for example to ensure internal files are not backed up to iCloud on iOS



```
void AddSkipBackupAttribute(string filename)
{
    if (File.Exists(filename)) {
        // Do not backup to iCloud
        NSFileManager.SetSkipBackupAttribute(filename, true);
    }
}
```

Can tell iOS to *not* backup a file in the documents folder to iCloud

File Formats

All three platforms support text, binary, XML and JSON formats – use the one that makes sense for your data style

```
using System.Xml.Linq;
...
public IEnumerable<string> LoadSurvey(StreamReader file)
{
    XmlDocument doc = XmlDocument.Load(file);
    return (from item in doc.Root.Descendants("question")
            select (string) item.Attribute("text")).ToList();
}
```

Recall: Xamarin.Forms architecture

Xamarin.Forms applications have two projects (shared + platform) that work together to provide the logic + UI for each executable



- PCL *shared* across all platforms
- limited access to .NET APIs
- want most of our code here

- one project per platform
- code is *not* shared
- full access to .NET APIs
- platform-specific code goes here

I/O in Portable Class Libraries

PCLs are limited to APIs which are available on all targeted frameworks, which impacts the classes we can use for I/O

```
public IEnumerable<string> LoadSurvey(string filename)
{
    var reader = new System.IO.StreamReader(filename); ❌
    ...
}
```

None of the available constructors on **StreamReader** takes a **string**

public StreamReader(
 System.IO.Stream stream
)
Parameter
stream: The stream to be read.
Summary
Initializes a new instance of the *StreamReader* class for the specified stream.

Types of data

The way you are using the data will be the determining factor in where it must be placed



Read-only, pre-supplied data

Data which is supplied with the application that never changes; can be bundled into the app itself

Types of data

The way you are using the data will be the determining factor in where it must be placed



Read-only, pre-supplied data

Locally created data

Original data created completely by the user of the application – must be stored in a writable location

Types of data

The way you are using the data will be the determining factor in where it must be placed

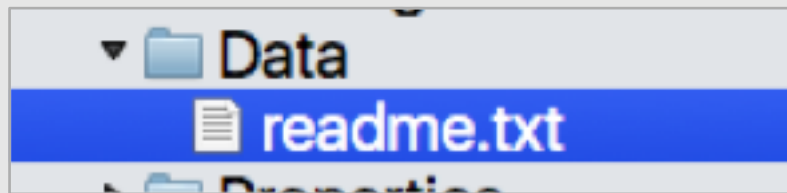
Read-only, pre-supplied data

Data which is pre-created by the developer, but then updated or refreshed by the user

Locally updated, pre-supplied data

Shipping read-only data with your app

Files added to your PCL projects can be added to the assembly if they have the build type "EmbeddedResource"

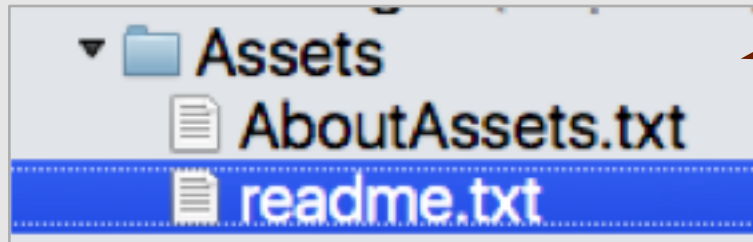


Build	
Build action	EmbeddedResource

```
Stream readme = typeof(eXam.App).GetTypeInfo().Assembly
    .GetManifestResourceStream("eXam.Data.readme.txt");
using (StreamReader sr = new StreamReader(readme))
{
    this.ReadmeText = sr.ReadToEnd();
}
```


Shipping read-only data with your app

Files added to your **native app projects** are included as part of the application and copied into the app sandbox when the app is installed



Files should be placed into existing **Assets** folder and have their build type set to **AndroidAsset**

Build	
Build action	AndroidAsset

Accessing bundled Android files

Android files can be loaded using the **Assets** property in your **Activity**

```
public partial class MainActivity
{
    protected override void onCreate(Bundle bundle) {
        ...
        string readmeText;
        using (StreamReader sr = new StreamReader(this.Assets.Open("readme.txt")))
        {
            readmeText = sr.ReadToEnd();
        }

        LoadApplication(new App { ReadmeText = readmeText });
    }
}
```



Passing data from native > cross-platform

Recall that native projects have references to the PCL; this allows the native projects to "push" or "inject" data into the shared code



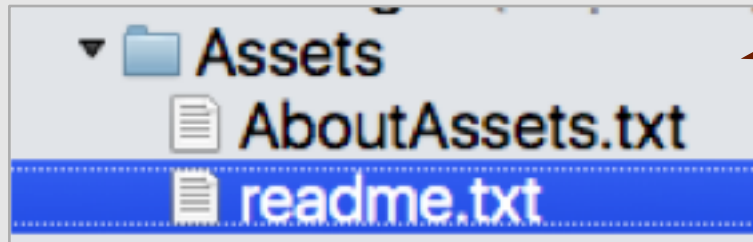
```
LoadApplication(new App() { ReadmeText = readmeText });
```

```
public class App : Application
{
    public string ReadmeText { get; set; }
    ...
}
```

PCL

Shipping read-only data with your app

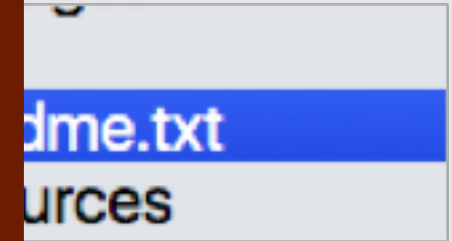
Files added to your **native app projects** are included as part of the application and copied into the app sandbox when the app is installed



Build	
Build action	AndroidAsset

Files can be placed anywhere in the project, but must use the build action "**BundleResource**"

iOS



Build	
Build action	BundleResource

Accessing bundled files

iOS files can be loaded using **System.IO** classes with the case-sensitive path and filename

```
public partial class AppDelegate
{
    public override bool FinishedLaunching( ... ) {
        ...

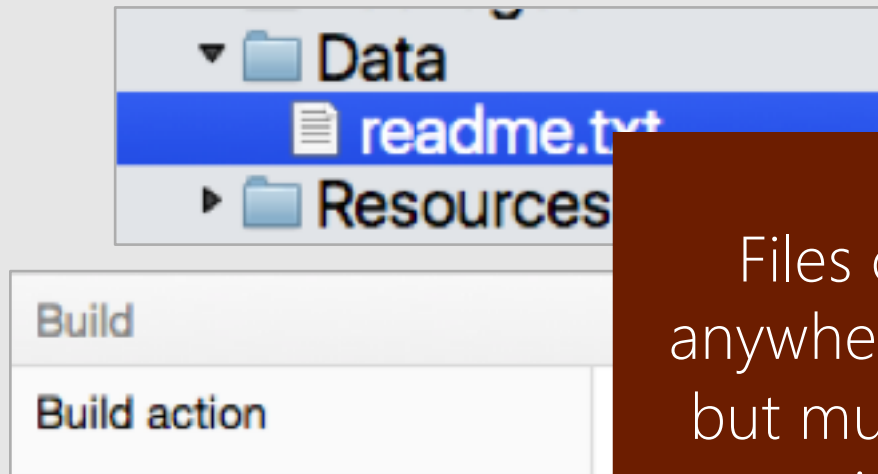
        string readmeText = System.IO.File.ReadAllText("Data/readme.txt");

        LoadApplication(new App() { ReadmeText = readmeText });
        return base.FinishedLaunching(app, options);
    }
}
```

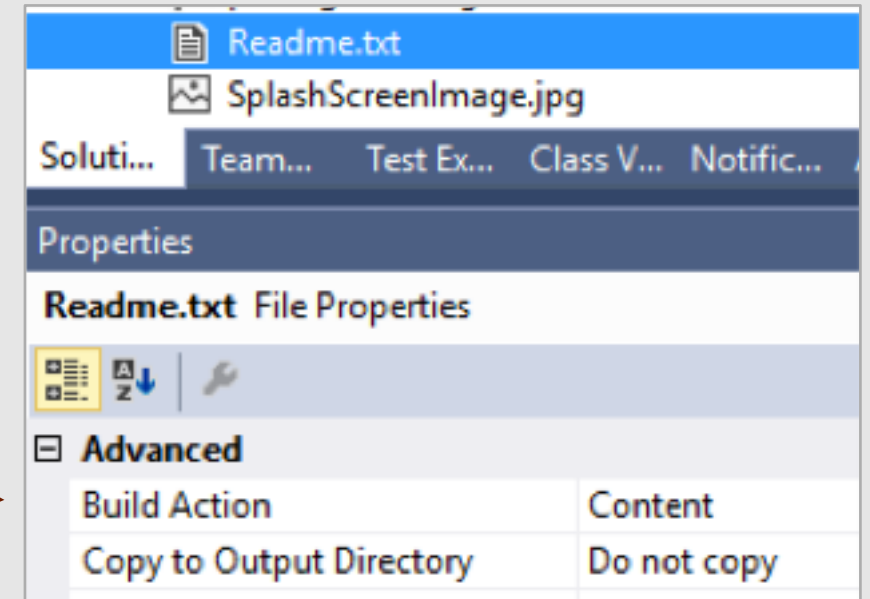
Shipping read-only data with your app

Files added to your **native app projects** are included as part of the application and copied into the app sandbox when the app is installed

iOS



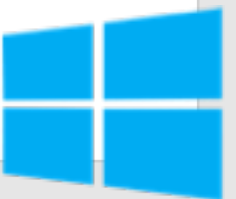
Files can be placed anywhere in the project, but must use the build action "Content"



Accessing bundled files

Windows files can be loaded using WinRT and **System.IO** classes, be aware that API is slightly different than full .NET 4.5

```
public partial class MainPage
{
    ...
    public async void LoadReadme(App theApp)
    {
        using (StreamReader sr = new StreamReader("readme.txt"))
        {
            app.ReadmeText = await sr.ReadToEndAsync();
        }
    }
}
```



Exercise #8

Load game data

Writable file locations

The recommended location for local data files which can be written to differs across platforms

Android	<code><AppHome>/files</code>
iOS	<code><AppHome>/Library/[subdirectory]</code>
Windows Phone	<code><AppHome>\local</code>



These locations are common, but other options are available (e.g. Android has a database folder)

Folder path [.NET]

Can use .NET APIs to get the full path to the application storage folder



```
// <AppHome>/files  
string path = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
```

iOS

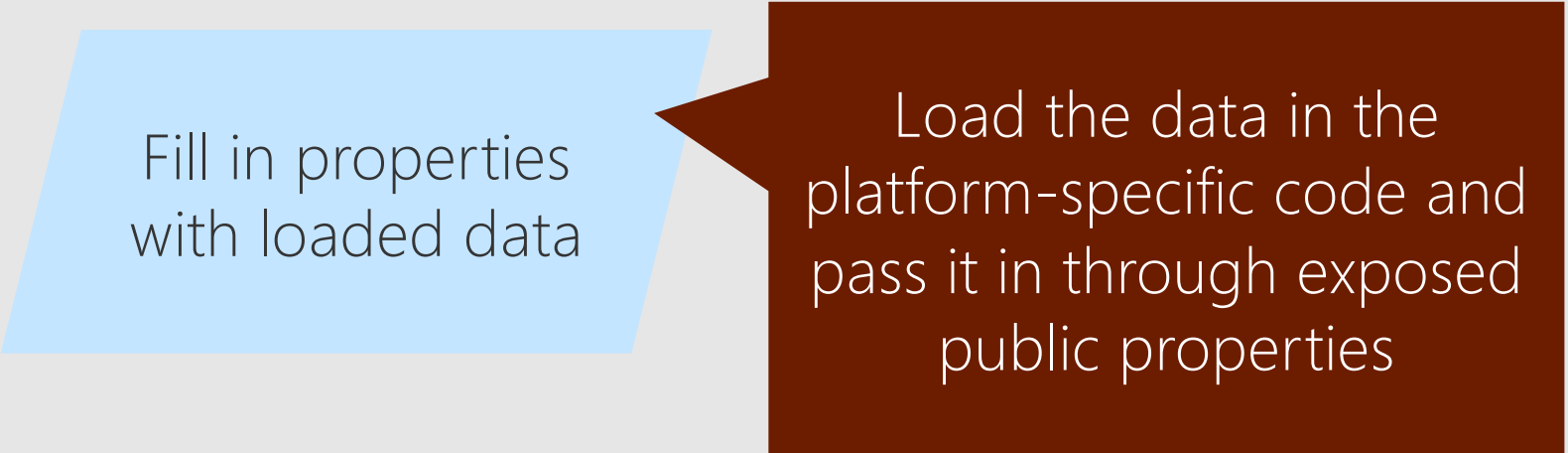
```
// <AppHome>/Documents  
string docFolder = Environment.GetFolderPath(Environment.SpecialFolder.Personal);  
// to meet Apple's iCloud terms, content that is not generated by the user  
// should be placed in the /Library folder or a subdirectory inside it  
string libFolder = System.IO.Path.Combine(docFolder, "..", "Library");
```



```
// <AppHome>\local  
string path = Windows.Storage.ApplicationData.Current.LocalFolder.Path;
```

Platform-specific code strategies

Several approaches you can take to passing data between the platform-specific code and the shared (PCL) code



Fill in properties
with loaded data

Load the data in the
platform-specific code and
pass it in through exposed
public properties

Platform-specific code strategies

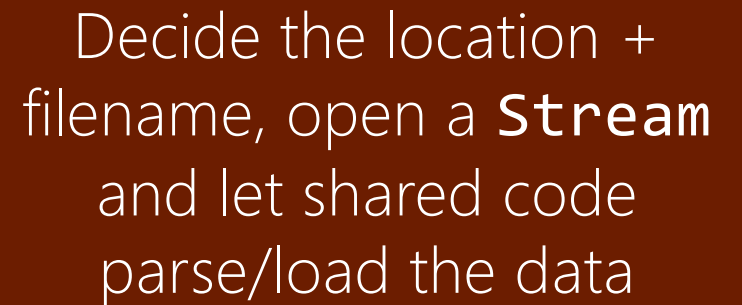
Several approaches you can take to passing data between the platform-specific code and the shared (PCL) code



Fill in properties
with loaded data



Open and pass
streams to PCL



Decide the location +
filename, open a **Stream**
and let shared code
parse/load the data

Platform-specific code strategies

Several approaches you can take to passing data between the platform-specific code and the shared (PCL) code

Fill in properties
with loaded data

Can use an abstraction such as
an interface or an event and
provide an implementation of
that abstraction in the
platform-specific project(s)

Design higher-
level abstractions

Platform Abstractions

Complex requirements can be described by an abstraction that is defined in the PCL

```
public interface IDialer
{
    bool MakeCall(string number);
}
```

PCL

Shared code defines **IDialer** interface to **represent required functionality** – this is what the PCL uses to get to the API

iOS PhoneDialerIOS



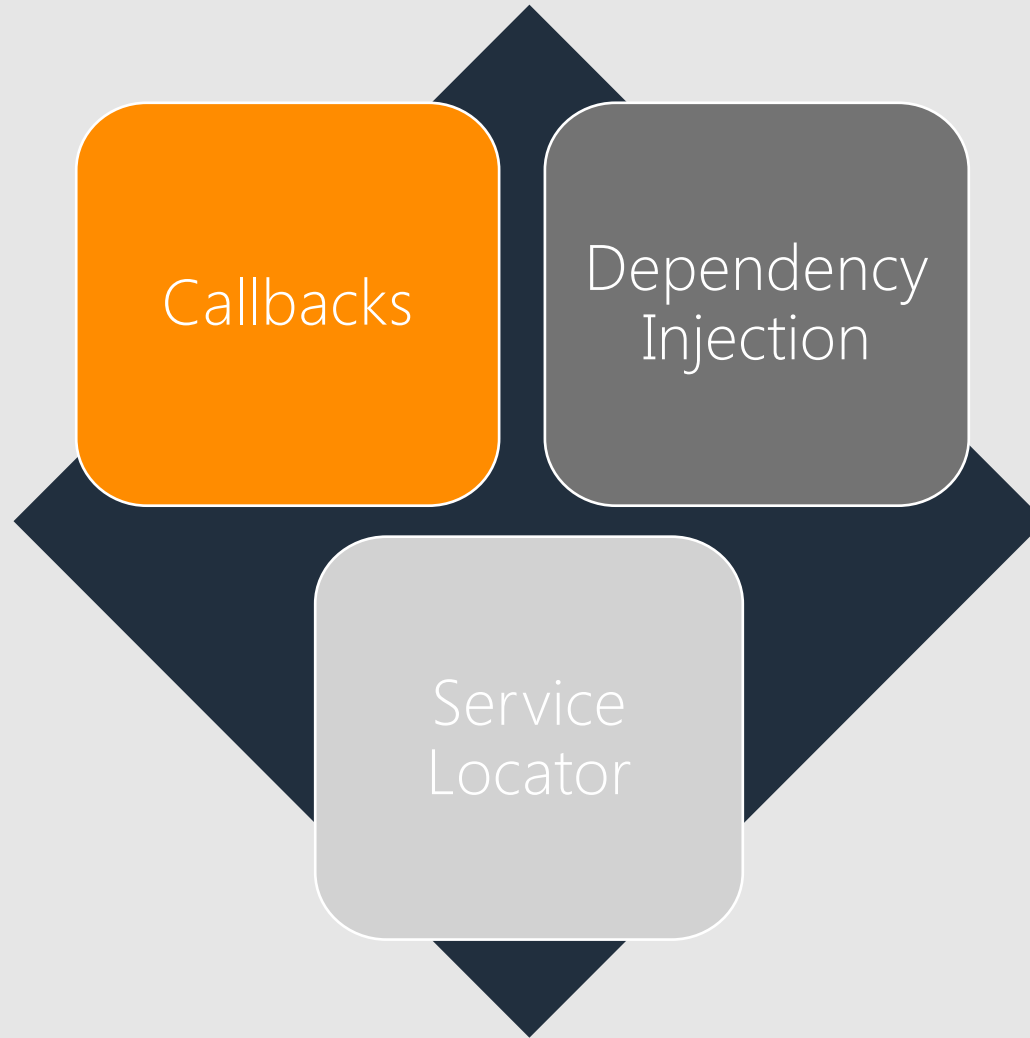
PhoneDialerDroid



PhoneDialerWin

Platform projects **implement the shared dialer interface** using the platform-specific APIs

Platform-specific code strategies



Callbacks


PCLs can **expose events or delegates** to request extensibility from the platform code, particularly effective if platform-specific requirements are small


```
public class Dialer
{
    public static Func<IDialer> Service;

    bool MakeCall(string number) {
        if (Service.MakeCall(number)) {
            ...
        }
    }
}
```

PCL

```
Dialer.Service = () =>
    new PhoneDialeriOS(); iOS
```

```
Dialer.Service = () =>
    new PhoneDialerDroid(); 
```

```
Dialer.Service = () =>
    new PhoneDialerWin(); 
```


Callbacks

PCLs can **expose events or delegates** to request extensibility from the platform code, particularly effective if platform-specific requirements are small

```
public class Dialer
{
    public static
        Func<string,bool> MakeCallImpl;

    bool MakeCall(string number) {
        if (MakeCallImpl(number)) {
            ...
        }
    }
}
```

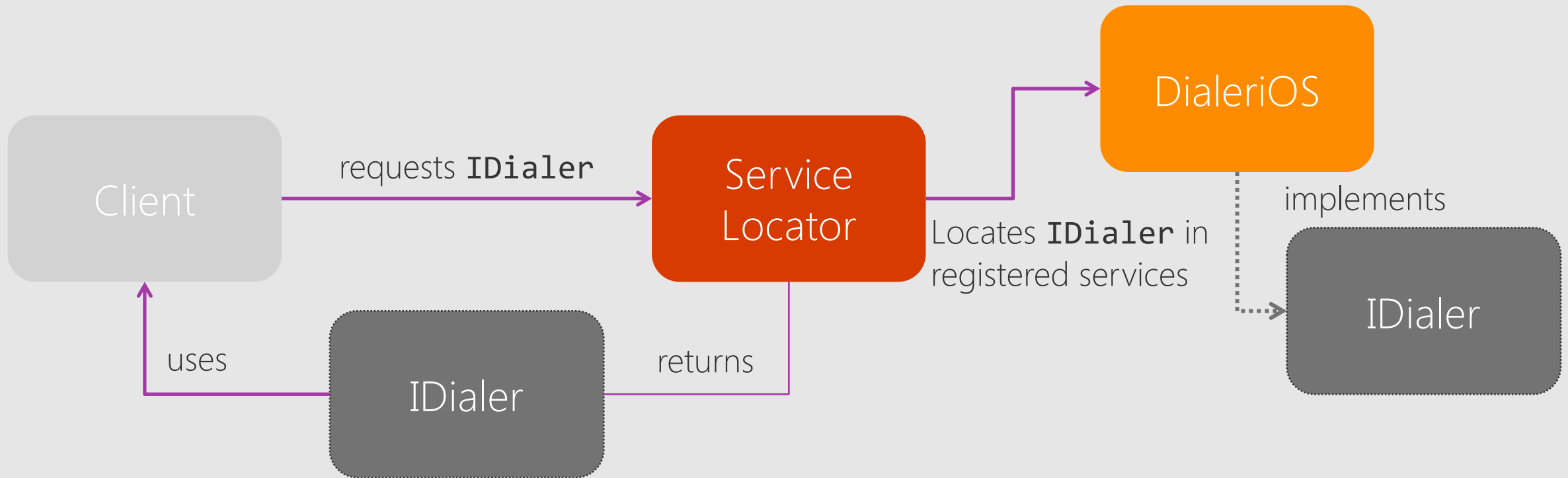
```
Dialer.MakeCallImpl = number =>
{
    return UIApplication
        .SharedApplication
        .OpenUrl(new NSURL(
            "tel:" + number));
}
```

iOS

PCL

What is a Service Locator?

Service Locator pattern uses a container that maps abstractions (interfaces) to concrete, registered types – client then uses locator to find dependencies



Service Locator in Xamarin.Forms

Xamarin.Forms includes a *service locator* called **DependencyService** which can be used to register platform-specific implementations

Define implementation as class that implements abstraction

```
class DialeriOS : IDialer { ... }
```

iOS


Then register implementation with **DependencyService** using **assembly-level attribute** in platform-specific project

```
[assembly: Dependency(typeof(DialeriOS))]
```

Locating dependencies

Can then locate dependencies in any project by requesting the abstraction with the **DependencyService.Get<T>** method

```
IDialer dialer = DependencyService.Get<IDialer>();  
if (dialer != null) {  
    ...  
}
```



We request an **IDialer**, but we get back a real implementation that's registered by the platform-specific project!

Dependency Injection

Another option is to have the platform-specific code "inject" the dependency through some form of initialization code; this avoids the dependency against a global service locator

When platform code creates the PCL **Application** object, it passes required dependencies to the constructor

```
public class App : Application
{
    public App (IDialer dialer) { ... }
    ...
}
```

```
LoadApplication(new App(new PhoneDialerWin()));
```



Exercise #9

Cache game data

Summary

- Can use preferences, databases, or local files to store app data
- Filenames and locations are defined on a per-platform basis
- File I/O is limited to basic streams in shared code, must provide implementations in each platform-specific project
- Can use callbacks, dependency injection or service location to connect shared code to platform-specific code