Improve App Performance

Cross-Platform Performance

1. Use the Profiler

- Profiling is a technique for determining where code optimizations will have the greatest effect in reducing performance problems
- It measure, evaluate, and help find performance-related issues in an application.
- Avoid profiling an application in a simulator, as the simulator may distort the application performance

2. Release IDisposable Resources

- The IDisposable interface provides a mechanism for releasing resources.
- It provides a Dispose method that should be implemented to explicitly release resources.
- Typical unmanaged resources that require releasing include files, streams, and network connections.
- two approaches for achieving this:

1. By wrapping the IDisposable object in a using statement.
2. By wrapping the call to IDisposable.Dispose in a try/finally block.

3. Unsubscribe from Events

- To prevent memory leaks, events should be unsubscribed from before the subscriber object is disposed of
- As long as the publishing object holds subscriber reference, garbage collection will not reclaim the subscriber object memory.

4. Use Weak References to Prevent Immortal Objects

- In some situations it's possible to create strong reference cycles that could prevent objects from having their memory reclaimed by the garbage collector
- One way to prevent a cycle is to use a weak reference from the child to the parent

```
WeakReference<Container> weakParent;
```

5. Delay the Cost of Creating Objects

- Lazy initialization can be used to defer the creation of an object until it's first used.
- used to improve performance, avoid computation, and reduce memory requirements.
- Lazy class is used to define a lazy-initialized type

```
Lazy<double> data = new Lazy<double>(() =>
 {

 });
```

6. Implement Asynchronous Operations

- Unlike synchronous APIs, the asynchronous APIs ensure that the active execution thread never blocks the calling thread for a significant amount of time.
- Therefore, when calling an API from the UI thread, use the asynchronous API if it's available. This will keep the UI thread unblocked, which will help to improve the user's experience with the application.
- long running operations should be executed on a background thread, to avoid blocking the UI thread.
- using async and await keywords,long running operations can be executed asynchronously ,but not guarantee that the operation will run on a background thread.
- can be accomplished by passing the long running operation to Task.Run
- Long running operations should also support cancellation

7. Use the SGen Garbage Collector

- The two garbage collectors used by the Xamarin platform are:

1. SGen – This is a generational garbage collector and is the default garbage collector on the Xamarin platform.
2. Boehm – This is a conservative, non-generational garbage collector. It is the default garbage collector used for Xamarin.iOS applications that use the Classic API.

8. Reduce the Size of the Application

- Xamarin platform includes a linker as part of the build tools.
- At build time, it will perform static analysis of the application to determine which types, and members, are actually used by the application. It will then remove any unused types and methods from the application.
- linker provides three different settings to control its behavior

1. Don’t Link

- No unused types and methods will be removed by the linker.

2. Link Framework SDKs/SDK Assemblies Only

- This setting will only reduce the size of those assemblies that are shipped by Xamarin.
- User code will be unaffected.

3. Link All Assemblies

- This is a more aggressive optimization that will target the SDK assemblies and user code

- The following steps can be used to further reduce the application executable size:

1. Ensure that a Release build is produced.
2. Reduce the number of architectures that the application is built for, to avoid a FAT binary being produced.
3. Ensure that the LLVM compiler is being used, to generate a more optimized executable.
4. Reduce the application's managed code size.

9. Optimize Image Resources

- reduce the CPU usage and memory footprint by creating multiple resolution versions of stored images that are close to the predicted display sizes

10. Reduce the Application Activation Period

- All applications have an activation period, which is the time between when the application is started and when the application is ready to use.
- The activation period can be reduced by ensuring that required resources are packaged within the app, instead of being retrieved remotely
- app's activation logic should only perform work that's required to let the user start using the application.

11. Reduce Web Service Communication

- it's sensible to limit the bandwidth utilization between an application and a web service.
- One approach to reducing an application's bandwidth utilization is to compress data before transferring it over a network,need to consider trade off for compression.
- JSON is often the preferred format for mobile applications.
- Data retrieved from the web service should be cached locally, with the cached data being utilized rather than repeatedly retrieved from the web service.

Optimizing App Performance with Xamarin.Forms

1. Enable the XAML compiler

- as XAML can be optionally compiled directly into intermediate language (IL) with the XAML compiler (XAMLC).

- enabling XAML compilation at the assembly level

```
using Xamarin.Forms.Xaml;
...
 [assembly: XamlCompilation (XamlCompilationOptions.Compile)]
 namespace DemoApp
 {
 ...
 }
```

- At Class Level

```
using Xamarin.Forms.Xaml;
...
[XamlCompilation (XamlCompilationOptions.Compile)]
public class HomePage : ContentPage
 {
```

```
  ...
}
```

- Benifits:

1. compile-time checking of XAML, errors are known
2. removes some of the load and instantiation time for XAML element
3. helps to reduce the file size of the final assembly by no longer including .xaml files

2. Use compiled bindings

- Compiled bindings resolving binding expressions at compile time, rather than at runtime with reflection.
- Compiling a binding expression is 8-20 times quicker than using a classic binding,so improve data binding performance
- The process for using compiled bindings is to:

1. Enable XAML compilation.
2. Set an x:DataType attribute on a Page or Layout

- To use compiled bindings, the x:DataType attribute must be set to

1. a string literal or
2. a type using the x:Type markup extension.

```
<ContentPage xmlns:viewModel="clr-namespace:BindableLayout.ViewModel"
x:DataType="viewModel:PlatformsViewModel"  >

<!--x:Datatype can be set on Page as well as Layout -->
```

- for DataTemplate we need to set the x:Datatype value to the model template is using.

```
 <DataTemplate x:DataType="models:DataModel">
 </DataTemplate>
```

3. Reduce unnecessary bindings

- Don't use bindings for content that can easily be set statically

4. Use fast renderers

- Fast renderers reduce the inflation and rendering costs of Xamarin.Forms controls on Android by flattening the resulting native control hierarchy
- further improves performance by creating fewer objects, which in turns results in a less complex visual tree, and less memory use.

- From Xamarin.Forms 4.0 onwards, all applications targeting FormsAppCompatActivity will use these fast renderers by default
- Custom renderers can be created for fast renderers using the same approach as used for the legacy renderers

- Backward compatibility

- Fast renderers can be overridden with the following approaches

1. Enabling the legacy renderers by adding code to your MainActivity class before calling Forms.Init:

```
Forms.SetFlags("UseLegacyRenderers");
```

2. Using custom renderers that target the legacy renderers.

5. Enable startup tracing on Android

- Ahead of Time (AOT) compilation on Android minimizes Just in Time (JIT) application startup overhead and memory usage, at the cost of creating a much larger APK
- alternative is to use startup tracing, which provides a trade-off between Android APK size and startup time, when compared to conventional AOT compilation.
- , startup tracing compiles only the set of managed methods that represent the most expensive parts of application startup in a blank Xamarin.Forms application.
- This approach results in a reduced APK size, when compared to conventional AOT compilation, while still providing similar startup improvements.

6. Enable layout compression

- Layout compression removes specified layouts from the visual tree, in an attempt to improve page rendering performance.
- Layout begins at the top of the visual tree with a page, and it proceeds through all branches of the visual tree to encompass every visual element on a page.
- Elements that are parents to other elements are responsible for sizing and positioning their children relative to themselves.
- Invalidation is the process by which a change in an element on a page triggers a new layout cycle
- The result of the layout process is a hierarchy of native controls. However, this hierarchy includes additional container renderers and wrappers for platform renderers, further inflating the view hierarchy nesting
-

- Layout compression aims to flatten the view nesting by removing specified layouts from the visual tree, which can improve page-rendering performance.
- to enable layout compression

```
<StackLayout CompressedLayout.IsHeadless="true">
...
```

```
    </StackLayout>
```

- Note:

- Since layout compression removes a layout from the visual tree, it's not suitable for layouts that have a visual appearance, or that obtain touch input.
- Therefore, layouts that set VisualElement properties
- such as BackgroundColor, IsVisible, Rotation, Scale, TranslationX and TranslationY or that accept gestures, are not candidates for layout compression.
- if used on them , they will fail silently

7. Choose the correct layout

- don't attempt to reproduce the appearance of a specific layout by using combinations of other layouts, as this results in unnecessary layout calculations being performed.
- A layout that's capable of displaying multiple children, but that only has a single child, is wasteful.

8. Optimize layout performance

- Reduce the depth of layout hierarchies by specifying Margin property values, allowing the creation of layouts with fewer wrapping views.
- When using a Grid, try to ensure that as few rows and columns as possible are set to Auto size. , instead use a proportional amount of space with the GridUnitType.Star enumeration value
- Don't set the VerticalOptions and HorizontalOptions properties of a layout unless required.
- Avoid using a RelativeLayout whenever possible.
- When using an AbsoluteLayout, avoid using the AbsoluteLayout.AutoSize property whenever possible.
- When using a StackLayout, ensure that only one child is set to LayoutOptions.Expands.
- Avoid calling any of the methods of the Layout class, as they result in expensive layout calculations being performed, the desired layout behavior can be obtained by setting the TranslationX and TranslationY properties.
- Don't update any Label instances more frequently than required
- Don't set the Label.VerticalTextAlignment property unless required
- Set the LineBreakMode of any Label instances to NoWrap whenever possible

9. Use asynchronous programming

- In .NET, the Task-based Asynchronous Pattern (TAP) is the recommended design pattern for asynchronous operations
- using it, overall responsiveness of your application can be enhanced
- ***Guidelines to follow***
- Use the Task.WhenAll method to asynchronously wait for multiple asynchronous operations to finish
- Use the Task.WhenAny method to asynchronously wait for one of multiple asynchronous operations to finish
- Use the Task.Delay method to produce a Task object that finishes after the specified time
- Execute intensive synchronous CPU operations on the thread pool with the Task.Run method.
- Avoid trying to create asynchronous constructors.

- Use the lazy task pattern to avoid waiting for asynchronous operations to complete during application startup.
- Return a Task object, instead of returning an awaited Task object,as less context switching.
- Use the Task Parallel Library (TPL) Dataflow library in scenarios such as processing data as it becomes available, or when you have multiple operations that must communicate with each other asynchronously

- **For UI**

- Call an asynchronous version of an API, if it's available. help to improve the user's experience as UI thread is unblocked.
- Update UI elements with data from asynchronous operations on the UI thread, to avoid exceptions being thrown.
- Any control properties that are updated via data binding will be automatically marshaled to the UI thread.

- **For Error**

- Avoid creating async void methods, and instead create async Task methods. These enable easier error-handling, composability, and testability.except for async event handler.
- Use the ConfigureAwait method whenever possible, to create context-free code. Context-free code has better performance for mobile applications and is a useful technique for avoiding deadlock when working with a partially asynchronous codebase.
- By using ConfigureAwait, you enable a small amount of parallelism: Some asynchronous code can run in parallel with the GUI thread instead of constantly badgering it with bits of work to do.

10. Choose a dependency injection container carefully

- Dependency injection containers introduce additional performance constraints into mobile applications
- Registering and resolving types with a container has a performance cost
- dependency injection can be made more performant by implementing it manually using factories,like in prism.

11. Create Shell applications

- Xamarin.Forms Shell reduces the complexity of mobile application development by providing the fundamental features that most mobile applications require, including:

1. A single place to describe the visual hierarchy of an application.
2. A common navigation user experience.
3. A URI-based navigation scheme that permits navigation to any page in the application.
4. An integrated search handler.
5. also it has an increased rendering speed, and reduced memory consumption.
6. Existing applications can adopt Shell

- Xamarin.Forms Shell applications provide an opinionated navigation experience based on flyouts and tabs.
- Shell applications help to avoid a poor startup experience, because pages are created on demand in response to navigation rather than at application startup

-

12. Use CollectionView instead of ListView

- CollectionView is a view for presenting lists of data using different layout specifications.
- provides a more flexible, and performant alternative to ListView.

13. Optimize ListView performance

- there are a number of user experiences that should be optimized:

1. Initialization
2. Scrolling
3. Interaction

14. Optimize image resources

1. if an application is displaying an image by reading its data from a stream

- ensure that stream is created only when it's required, and ensure that the stream is released when it's no longer required
- like in Page Appearing/Disappearing method.

2. downloading an image for display with the ImageSource.FromUri method, cache the downloaded image by ensuring that the UriImageSource.CachingEnabled property is set to true.

15. Reduce the visual tree size

- Reducing the number of elements on a page will make the page render faster.
- two main techniques for achieving this

1. hide elements that aren't visible.

- set IsVisible property of element to true or false , to determines whether the element should be part of the visual tree or not.

2. remove unnecessary elements

16. Reduce the application resource dictionary size

- Any resources that are used throughout the application should be stored in the application's resource dictionary to avoid duplication
- help to reduce the amount of XAML that has to be parsed

17. Use the custom renderer pattern

- Xamarin.Forms renderer classes expose the OnElementChanged method, which is called when a Xamarin.Forms custom control is created to render the corresponding native control