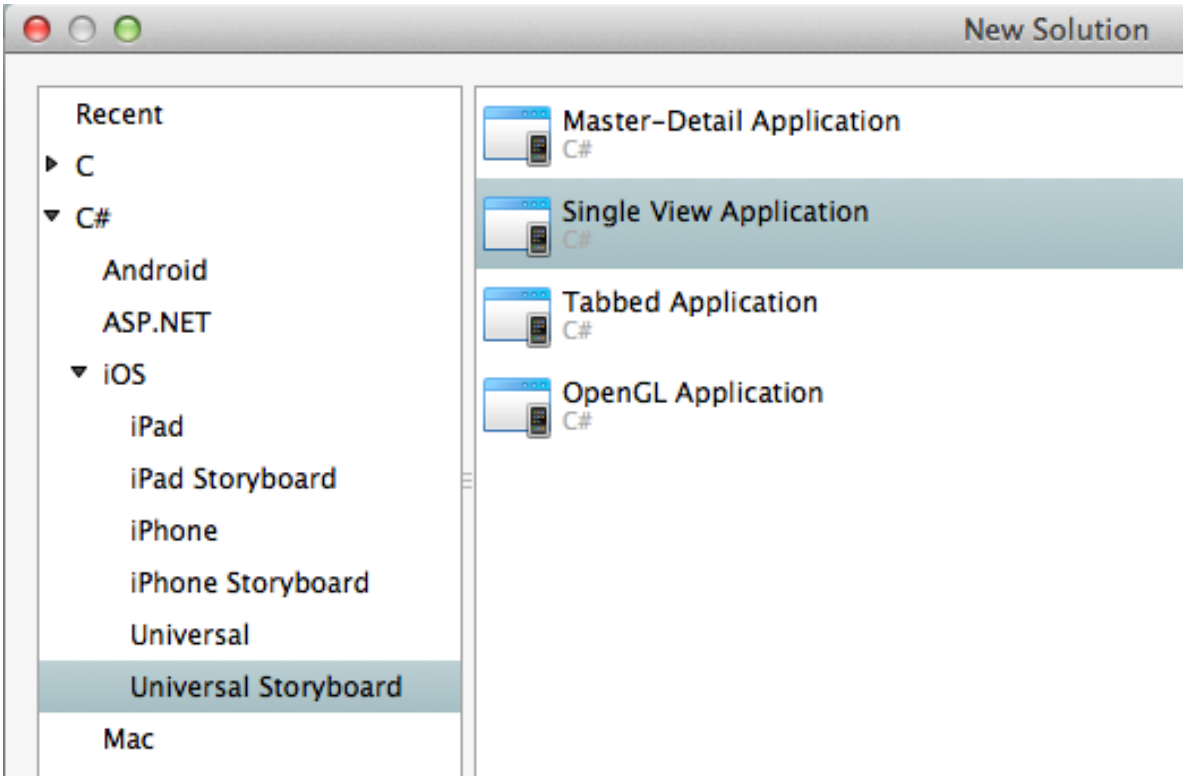
Tutorial: Creating a Viewer for Excel files

Note: The complete source for this tutorial is available as "FlexView" demo in the FlexCel distribution.

Step 1: Setting up the Application

Let's start by creating a new iOS single view application:



Give it a name (we will use FlexView) and save it.

Then you can go to the application properties, and set icons and the application name.

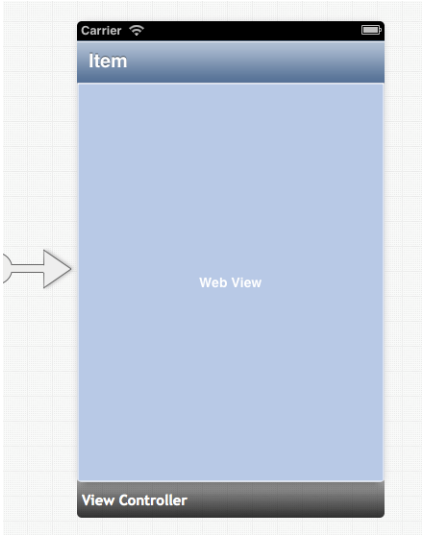
You might now try running the application, it should show as an empty form in the simulator or the device.

Step 2: Creating the user interface.

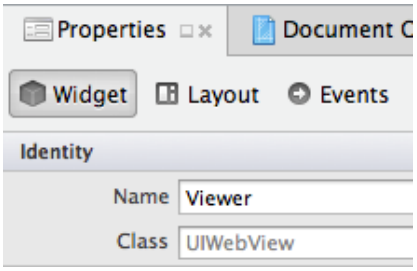
We want to display an Excel file into our application. For this, we will convert the file to pdf using FlexCel, and show the pdf output in a web browser.

Double click in the file MainStoryboard_iPhone.storyboard. Depending in your version of Xamarin Studio and in your preferences, the file might open in XCode or the Xamarin designer.
We'll be showing the Xamarin Designer here, but the XCode steps are similar.

In the designer, drop a WebView and a Toolbar. Adjust the anchors so the WebView resizes with the window:

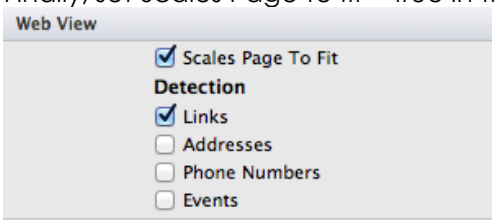


If using Xamarin Designer, Name the WebView Viewer:



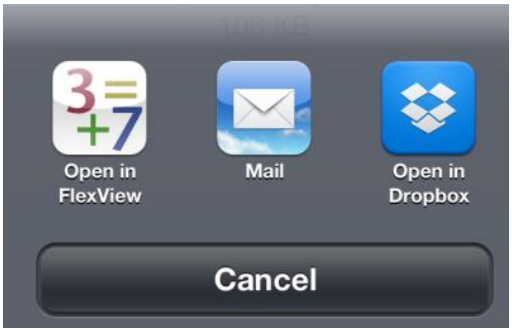
If using XCode, create an Outlet for the webview named "Viewer" by ctrl-dragging the webview to the assistant view.

Finally, set Scales Page to fit = true in the web properties to enable pinch to zoom:



Step 3: Registering the application with iOS as xls and xlsx handler.

The next step is to tell iOS that our application can handle xls and xlsx files. This way, when other app like for example mail wants to share an xls or xlsx file, our application will show in the list of available options:



To register our app, we need to change the file Info.plist. This can be done directly from the Xamarin Studio Info.plist editor, but for this example we'll just open the new generated Info.plist with a text editor, and paste the following text before the last </dict> entry:

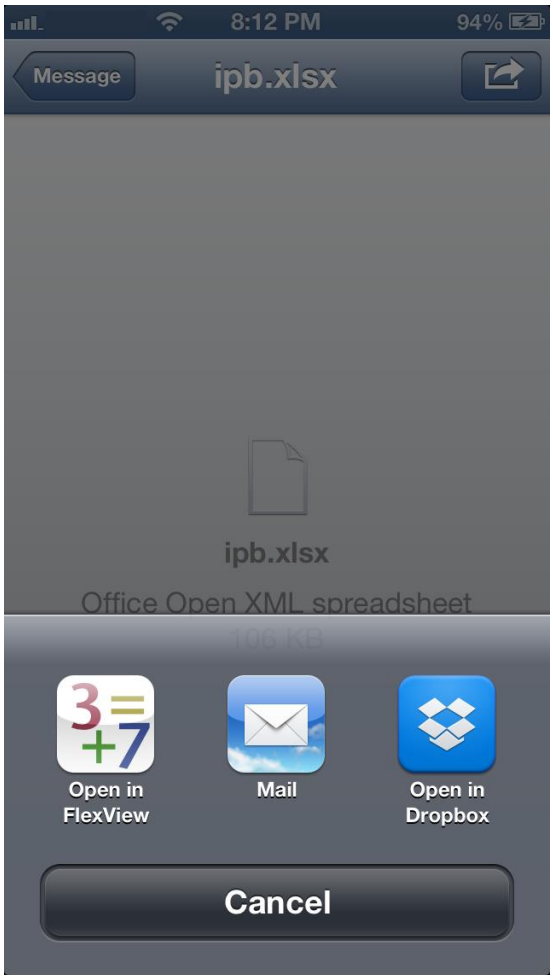
```
<key>CFBundleDocumentTypes</key>
<array>
    <dict>
        <key>CFBundleTypeName</key>
        <string>Excel document</string>
        <key>CFBundleTypeRole</key>
        <string>Editor</string>
        <key>LSHandlerRank</key>
        <string>Owner</string>
        <key>LSItemContentTypes</key>
        <array>
            <string>com.microsoft.excel.xls</string>
            <string>com.tms.flexcel.xlsx</string>

     <string>org.openxmlformats.spreadsheetml.sheet</string>
        </array>
    </dict>
</array>


<key>UTExportedTypeDeclarations</key>
 <array>
<dict>
    <key>UTTypeDescription</key>
    <string>Excel xlsx document</string>
    <key>UTTypeTagSpecification</key>
    <dict>
        <key>public.filename-extension</key>
```

```
            <string>xlsx</string>
            <key>public.mime-type</key>
            <string>application/vnd.openxmlformats-
officedocument.spreadsheetml.sheet</string>
        </dict>
        <key>UTTypeConformsTo</key>
        <array>
            <string>public.data</string>
        </array>
        <key>UTTypeIdentifier</key>
        <string>com.tms.flexcel.xlsx</string>
    </dict>
    </array>
```

Once you have done this, if you run the application and have for example an email with an xls or xlsx file, you should see "FlexView" in the list of available applications:



Step 4: Opening the file when another application sends it.

If you tried the application after step 3, and pressed the "Open in FlexView" button, you will notice that FlexView starts, but the previewer is still empty. It won't show the file that the other application sent.

What happens when you press the "Open in FlexView" button is that iOS will copy the file in the "Documents/Inbox" private folder of FlexView, and send an OpenURL event to our app. We need to handle this event, and use it to load the file in the preview.

Open AppDelegate.cs in Xamarin Studio, and write the following code:

```
public override bool OpenUrl(UIApplication application, NSUrl url,
      string sourceApplication, NSObject annotation)
{
      return FlexViewViewController.Open(url);
}
```

Note that you might just write "override" inside the AppDelegate class, an Xamarin Studio will show you a list of possible methods to override.

Note: In iOS, we are going to get the URL of the file, not the filename. For example, the URL could be:

'file://localhost/private/var/mobile/Applications/9D16227A-CB01-465D-B8F4-AC43D70C8461/Documents/Inbox/test.xlsx'

And the actual filename would be: '/private/var/mobile/Applications/9D16227A-CB01-465D-B8F4-AC43D70C8461/Documents/Inbox/test.xlsx'

But while iOS methods can normally use an URL or a path, C# FileStream expects a path. This is why we need to convert the URL to a path, using the **url.Path**.

So now it is the time to do the actual work. We need to write the FlexCelViewViewController.Open(url) mehtod, that will convert the file to pdf and display it on the browser.

For this, open the file FlexCelViewViewController.cs, and write the following uses at the start:

```
using System;
using System.Drawing;
using MonoTouch.Foundation;
using MonoTouch.UIKit;
using FlexCel.Render;
using FlexCel.XlsAdapter;
using System.IO;
```

And then type the following code inside the class:

```
NSUrl XlsUrl;
string PdfPath;

public bool Open(NSUrl url)
{
    XlsUrl = url;
    return Refresh();
}
private bool Refresh()
{
    try
    {
        XlsFile xls = new XlsFile(XlsUrl.Path);
        PdfPath = Path.Combine(
Environment.GetFolderPath(Environment.SpecialFolder.InternetCache),
Path.ChangeExtension(Path.GetFileName(XlsUrl.Path), ".pdf"));

        using (FlexCelPdfExport pdf
                     = new FlexCelPdfExport(xls, true))
        {
            using (FileStream fs = new FileStream(PdfPath,
                   FileMode.Create))
            {
                pdf.Export(fs);
            }
        }
         Viewer.LoadRequest(new
                  NSUrlRequest(NSUrl.FromFilename(PdfPath)));
    }
    catch (Exception ex)
    {
        Viewer.LoadHtmlString("<html>Error opening " +
            System.Security.SecurityElement.Escape(
                Path.GetFileName(XlsUrl.Path))
            + "<br><br>"
            + System.Security.SecurityElement.Escape(
               ex.Message)
            + "</html>", null);

            return false;
    }
    return true;
}
```

If you run the application now and press “Open in FlexView” from another application, FlexView should start and display the file. You should be able to scroll and pinch to zoom.

Notes on temporary files and memory usage

Normaly when coding for a PC, you want to avaoid temporary files. In the code above, we could have directly exported the xls file to a memory stream, and then use that memory stream to create an NSData object and load it directly to the web browser.

But when coding for a phone, things are different. In this case, the memory of the device is limited, and the flash storage is fast and huge (it has to be able to store music and videos).

So, if memory is our constrain, it makes more sense to create a temporary file and read from there. While FlexCel will keep the spreadsheet in memory when opening, it won't keep the pdf file, which is generated on the fly. So saving this pdf file to a temp place can reduce the memory usage a lot.

Another thing to notice is that we should **not** write this temporary file to the /Documents folder, because this isn't user data, and it shouldn't be backed up by iTunes. That's why we write it to Library/Caches

And the last thing we are missing here, is to remove the file once the WebView loaded it. While this isn't strictly necessary (iOS will remove files in this cache when it needs space), it is a good practice.

We could remove the file in the Viewer.LoadFinished event, but since we plan to share the file to other apps in the next steps, we will keep it longer. So we will delete the old file before writing a new one.
In FlexCelViewViewController.cs, add the following method:

```
        private void RemoveOldPdf()
         {
             if (PdfPath != null)
             {
                 try
                 {
                     File.Delete(PdfPath);
                 }
                 catch
                 {
                     //do nothing, this was just a
                     //cache that will get deleted anyway.
                 }
                 PdfPath = null;
             }
         }
```

And call RemoveOldPdf() as the first line in the Refresh() mehtod we wrote above. Note that we could have also saved always to the same filename, so we wouldn't need to worry about deleting files at all. But when sharing the file, the filename would be lost.

Note: While we won't cover this here, another advantage of using a temporary file is that if your app gets killed by the OS, you can restore the state from the temporary file when restarted.

Step 5: Modifying the file

For tis step, we will be replacing all numbers in the file with random numbers and recalculating the file. While this doesn't make a lot of sense, it shows how you can modify a file.

In this particular case, the only difficulty is that we can't overwrite the original file at Documents/Inbox. It is readonly. So we will save it to the Caches folder. As tmpFlexCel.xls or tmpFlexCel.xlsx depending on the file format.

Saving it with the same name allows us to not care about deleting the temporary file because it will always be the same. But on the other hand, we need to store the original name of the file so the generated pdf file has the right filename.

So we'll introduce a new variable:

```
string XlsPath;
```

And we'll replace the old XlsUrl for XlsPath in all places where it isn't dealing with the pdf file.
Finally, we'll add a button and write this in the event handler:

```
partial void RandomizeClick(MonoTouch.UIKit.UIBarButtonItem sender)
{
    if (XlsUrl == null) return;

    XlsFile xls = new XlsFile(XlsPath, true);
    //We'll go through all the numeric cells and make them random
    Random rnd = new Random();

    for (int row = 1; row <= xls.RowCount; row++)
    {
        for (int colIndex = 1;
             colIndex < xls.ColCountInRow(row);
             colIndex++)
        {
            int XF = -1;
            object val = xls.GetCellValueIndexed(row, colIndex, ref
XF);
            if (val is double) xls.SetCellValue(row,
                    xls.ColFromIndex(row, colIndex), rnd.Next());
        }
    }
```

```
    //We can't save to the original file, we don't have permissions.
    XlsPath = Path.Combine(
        Environment.GetFolderPath(
             Environment.SpecialFolder.InternetCache),
        "tmpFlexCel" + Path.GetExtension(XlsUrl.Path));

    xls.Save(XlsPath);
    Refresh();
}
```

This code should do the replacement.

Step 6: Exporting the file to other applications

In step 4 we saw how to import a file from another application. In this step we are going to see how to do the opposite: How to export the file and make it available to other applications that handle xls or xlsx files. We will also see how to print the file.

Luckily, this isn't complex to do. Go to the UI designer, locate the "item" button in the toolbar, rename it "Share", and add the following code in the button event handler:

```
partial void ShareClick(MonoTouch.UIKit.UIBarButtonItem sender)
{
     if (PdfPath == null) return;
     UIDocumentInteractionController docController =
            new UIDocumentInteractionController();
     docController.Url = NSUrl.FromFilename(PdfPath);
     docController.PresentOpenInMenu(ShareButton, true);
}
```

This should show a dialog in your app to share the file with other apps.

And it might be what we want in many cases. But this dialog doesn't include the options to "Print", or "Mail", which might be interesting to show too.

To show the extended options, change the last line in the code above from PresentOpenInMenu to PresentOptionsMenu:

```
docController.PresentOptionsMenu(ShareButton, true);
```

Now you should see all options, and as this is a pdf file, even an option ot print it:

Step 7: Final touches

In this small tutorial we've gone from zero to a fully working Excel preview / pdf converter application. But for simplicity, we've conveniently "forgotten" about an interesting fact: Excel files can have more than one sheet.

Modifying the app so it allows you to change sheets isn't complex, on the FlexCel side you just need to use xls.SheetCount and xls.GetSheetName(sheetIndex) to get an array of the sheets, then use xls.ActiveSheetByName to set the sheet you want to display,

But while not complex, there is a lot of plumbing needed for that: we need to define a new view in the storyboard, we need to populate a table view with the items, and select the correct one when the user selects a new sheet. Sadly this is a lot of code, and would mean making this tutorial twice as big with little FlexCel code and a lot of UI code that you can get tutorials everywhere, and so we won't be showing how to do it here.