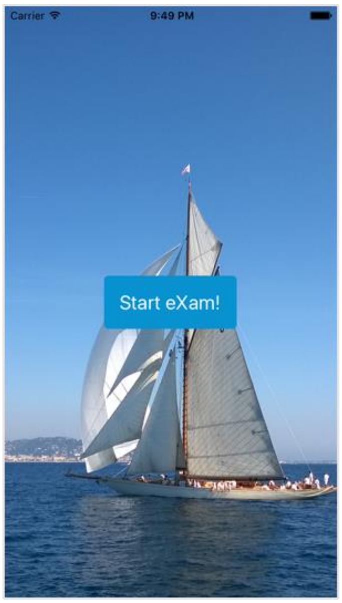
Ignite 2017 Pre-Day Training Lab Manual

Table of Contents

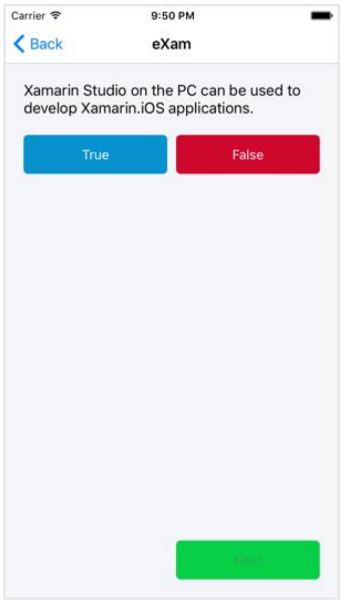
Exercises

You are going to build a simple quiz application in Xamarin Forms. The app will load a set of true/false questions from either a local file or a web URI and display them one at a time for the user to answer. It will track the user's answers and display their score at the end.

The app will have three pages:

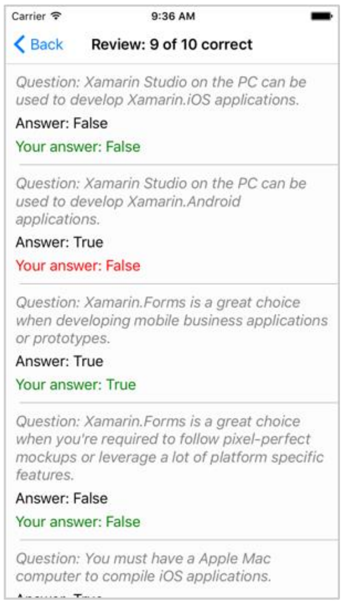


HomePage
shown at startup



Xamarin Studio on the PC can be used to develop Xamarin.iOS applications.

QuestionPage
displays a question



Question: Xamarin Studio on the PC can be used to develop Xamarin.iOS applications.
Answer: False
Your answer: False

Question: Xamarin Studio on the PC can be used to develop Xamarin.Android applications.
Answer: True
Your answer: False

Question: Xamarin.Forms is a great choice when developing mobile business applications or prototypes.
Answer: True
Your answer: True

Question: Xamarin.Forms is a great choice when you're required to follow pixel-perfect mockups or leverage a lot of platform specific features.
Answer: False
Your answer: False

Question: You must have a Apple Mac computer to compile iOS applications.

ReviewPage
summarizes results

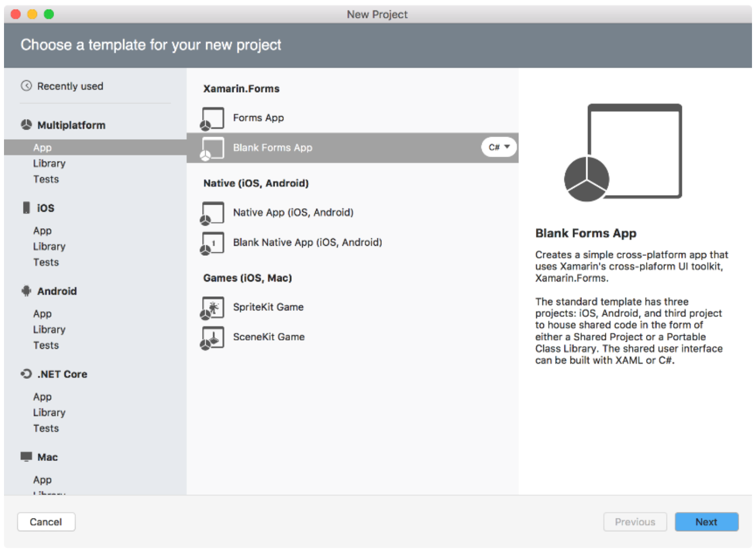
You will start from the beginning by creating a new project. You will code the parts of the app related to core Xamarin skills like UI creation and cross-platform techniques. Some of the platform-agnostic code, such as the game logic and model classes, will be supplied. This combination of your work and supplied code should allow you to practice key skills while creating a functional app in a short amount of time.

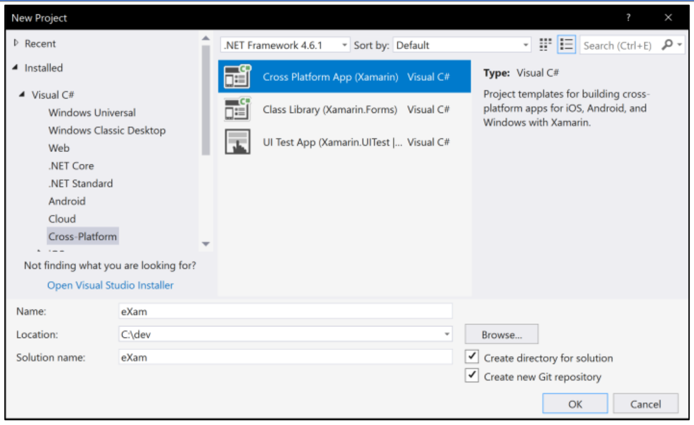
The supplied code and an image can be found in the **Assets** folder. Please make sure you have this folder before you begin.

1 Create the Xamarin.Forms project

1. Open either Visual Studio or Visual Studio for Mac

2. Create a new Xamarin.Forms PCL project named **eXam**:

VS for Mac	File → New Solution → Multiplatform → App → Blank Forms App *(not "Forms App")*  New Project Choose a template for your new project Recently used Multiplatform App Library Tests iOS App Library Tests Android App Library Tests .NET Core App Library Tests Mac App Xamarin.Forms Forms App Blank Forms App C# Native (iOS, Android) Native App (iOS, Android) Blank Native App (iOS, Android) Games (iOS, Mac) SpriteKit Game SceneKit Game Cancel Previous Next Blank Forms App Creates a simple cross-platform app that uses Xamarin's cross-plaform UI toolkit, Xamarin.Forms. The standard template has three projects: iOS, Android, and third project to house shared code in the form of either a Shared Project or a Portable Class Library. The shared user interface can be built with XAML or C#. Name: eXam Organization Identifier : leave as default Target Platforms : leave **iOS** and **Android** selected Shared Code : leave **Use Portable Class Library** selected XAML : leave **Use XAML for user interface files** selected Click **Next** Leave defaults selected and Click **Create**
Visual Studio	File → New → Project → Visual Studio C# → Cross-Platform → Cross-Platform App (Xamarin) → Name : eXam → Click OK *Warning : Windows has a finite path length. We recommend creating your solution close to the root of your drive.*



New Project

- Recent
- Installed
  - Visual C#
    - Windows Universal
    - Windows Classic Desktop
    - Web
    - .NET Core
    - .NET Standard
    - Android
    - Cloud
    - Cross-Platform

Not finding what you are looking for?
Open Visual Studio Installer

.NET Framework 4.6.1 | Sort by: Default | Search (Ctrl+E)

Cross Platform App (Xamarin)	Visual C#
Class Library (Xamarin.Forms)	Visual C#
UI Test App (Xamarin.UITest |...	Visual C#

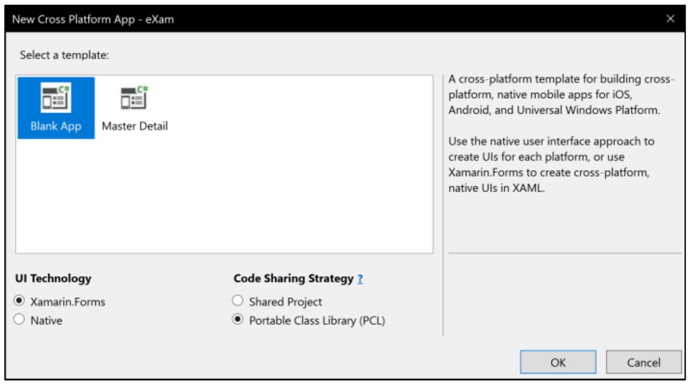
Type: Visual C#

Project templates for building cross-platform apps for iOS, Android, and Windows with Xamarin.

Name: eXam
Location: C:\dev
Solution name: eXam

[✓] Create directory for solution
[✓] Create new Git repository

In the Wizard, choose the Blank App template, with Xamarin.Forms as the UI Technology and Portable Class Library (PCL) selected



New Cross Platform App - eXam

Select a template:

Blank App | Master Detail

A cross-platform template for building cross-platform, native mobile apps for iOS, Android, and Universal Windows Platform.

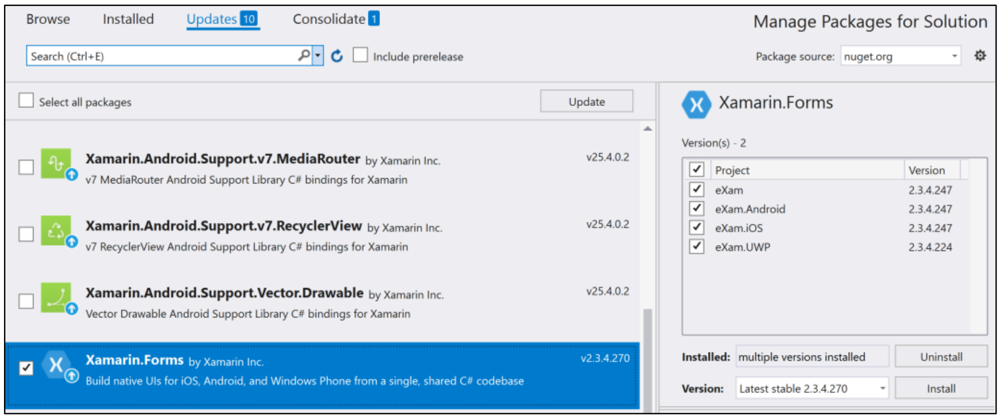
Use the native user interface approach to create UIs for each platform, or use Xamarin.Forms to create cross-platform, native UIs in XAML.

UI Technology
- ● Xamarin.Forms
- ○ Native

Code Sharing Strategy ?
- ○ Shared Project
- ● Portable Class Library (PCL)

For the UWP project, accept the default versioning.

3. Update just the Xamarin.Forms NuGet package to the latest stable version in each project.



Browse | Installed | Updates 10 | Consolidate 1 — Manage Packages for Solution

Search (Ctrl+E) | [ ] Include prerelease | Package source: nuget.org

[ ] Select all packages | Update

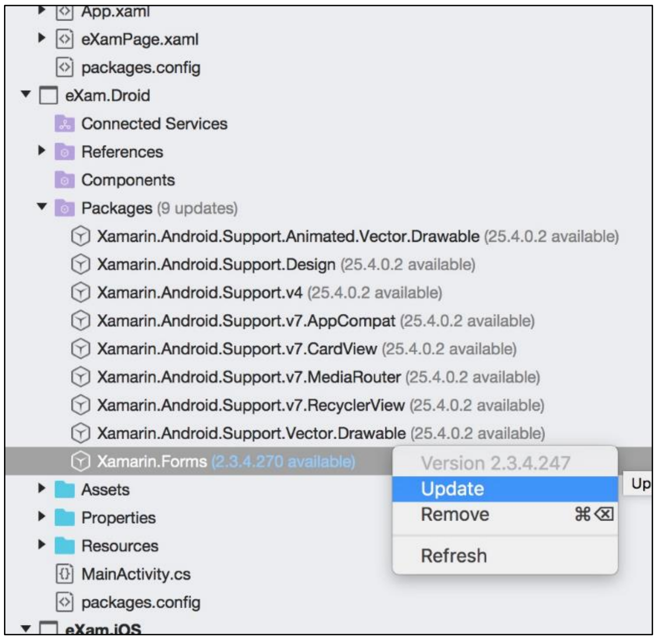
[ ]	Xamarin.Android.Support.v7.MediaRouter by Xamarin Inc. v7 MediaRouter Android Support Library C# bindings for Xamarin	v25.4.0.2
[ ]	Xamarin.Android.Support.v7.RecyclerView by Xamarin Inc. v7 RecyclerView Android Support Library C# bindings for Xamarin	v25.4.0.2
[ ]	Xamarin.Android.Support.Vector.Drawable by Xamarin Inc. Vector Drawable Android Support Library C# bindings for Xamarin	v25.4.0.2
[✓]	Xamarin.Forms by Xamarin Inc. Build native UIs for iOS, Android, and Windows Phone from a single, shared C# codebase	v2.3.4.270

Xamarin.Forms

Version(s) - 2

	Project	Version
✓	eXam	2.3.4.247
✓	eXam.Android	2.3.4.247
✓	eXam.iOS	2.3.4.247
✓	eXam.UWP	2.3.4.224

Installed: multiple versions installed | Uninstall
Version: Latest stable 2.3.4.270 | Install

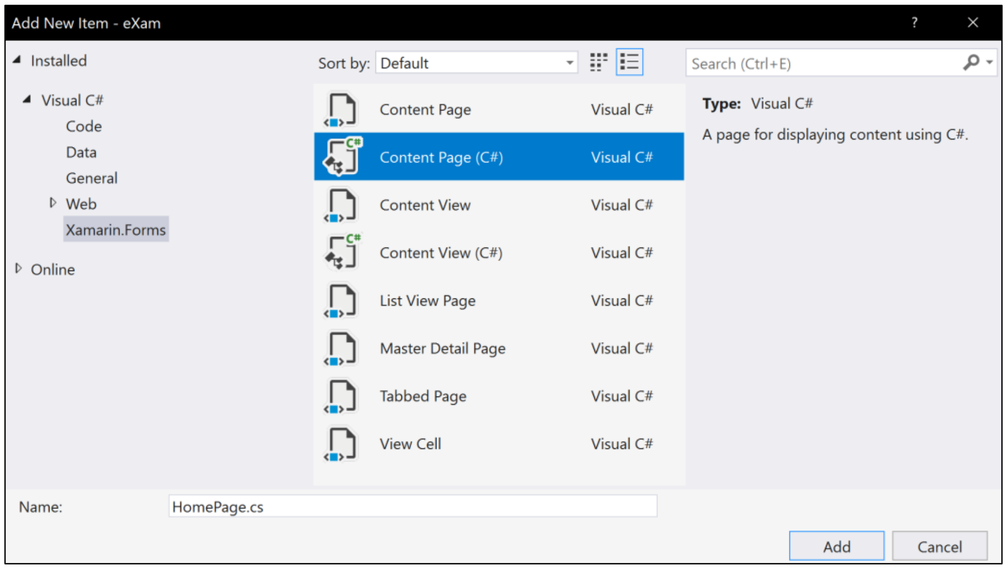

App.xaml
eXamPage.xaml
packages.config
eXam.Droid
Connected Services
References
Components
Packages (9 updates)
Xamarin.Android.Support.Animated.Vector.Drawable (25.4.0.2 available)
Xamarin.Android.Support.Design (25.4.0.2 available)
Xamarin.Android.Support.v4 (25.4.0.2 available)
Xamarin.Android.Support.v7.AppCompat (25.4.0.2 available)
Xamarin.Android.Support.v7.CardView (25.4.0.2 available)
Xamarin.Android.Support.v7.MediaRouter (25.4.0.2 available)
Xamarin.Android.Support.v7.RecyclerView (25.4.0.2 available)
Xamarin.Android.Support.Vector.Drawable (25.4.0.2 available)
Xamarin.Forms (2.3.4.270 available)
Assets
Properties
Resources
MainActivity.cs
packages.config
eXam.iOS

4. Run one or more of the platform-specific projects to verify your installation and setup are configured correctly. You can use either a simulator or a device.

2 Add a landing page programmatically

In this part, you will create the landing page and add a button centered in the page. This part is UI-only so the button will not have any behavior yet.

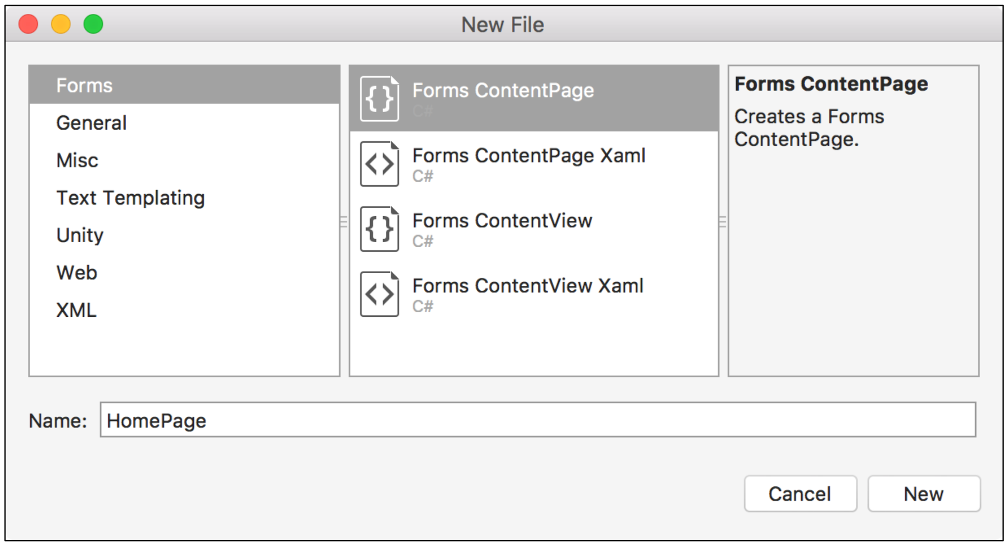
1. In the **eXam** PCL project, add a new Xamarin.Forms **Content Page (C#)** named **HomePage.cs**.



Type: Visual C#

A page for displaying content using C#.

Name: HomePage.cs



Forms ContentPage

Creates a Forms ContentPage.

Name: HomePage

2. Open the **HomePage.cs** file and locate the page's constructor, the next few steps will be done inside the constructor.
    a. Delete the default code from the constructor, leaving the empty constructor.
    b. Create an **AbsoluteLayout** and store it in a local variable.
    c. Assign the layout to the **Content** of the page.
    d. Create a **Button** and set its **Text** to "Start eXam!". Optionally, set the **TextColor**, **BackgroundColor** and **Font** to values of your choice.
    e. Add the button to the **Children** of the **AbsoluteLayout**. Optionally, pass positioning information to center the button on the page; use (0.5, 0.5, 150, 60) as the Rectangle parameter and **PositionProportional** for the **AbsoluteLayoutFlags**.

```
public HomePage()
{
    AbsoluteLayout layout = new AbsoluteLayout();

    this.Content = layout;

    Button button = new Button
    {
        Text = "Start eXam!",
        BackgroundColor = Color.LightCoral,
        Font = Font.SystemFontOfSize(20)
    };

    layout.Children.Add(
        view: button,
        bounds: new Rectangle(x: 0.5, y: 0.5, width: 150, height: 60),
        flags: AbsoluteLayoutFlags.PositionProportional);
}
```

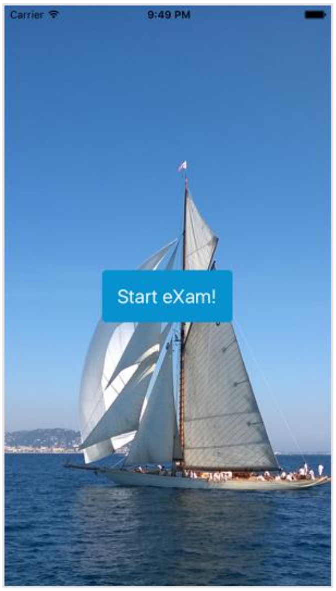
3. Open the Application class: **App.xaml.cs** .
    a. In the app's constructor, delete the default code and assign a new instance of **HomePage** to the app's **MainPage** property.

```
public App()
{
    InitializeComponent();

    MainPage = new HomePage();
}
```

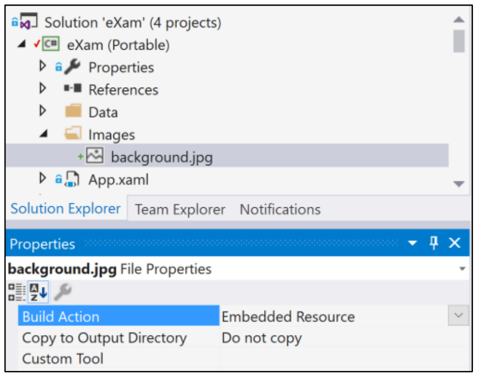
4. Run the application to test your work.

3 Display an image on the landing page

In this part, you will add a background image to **HomePage** (see below).

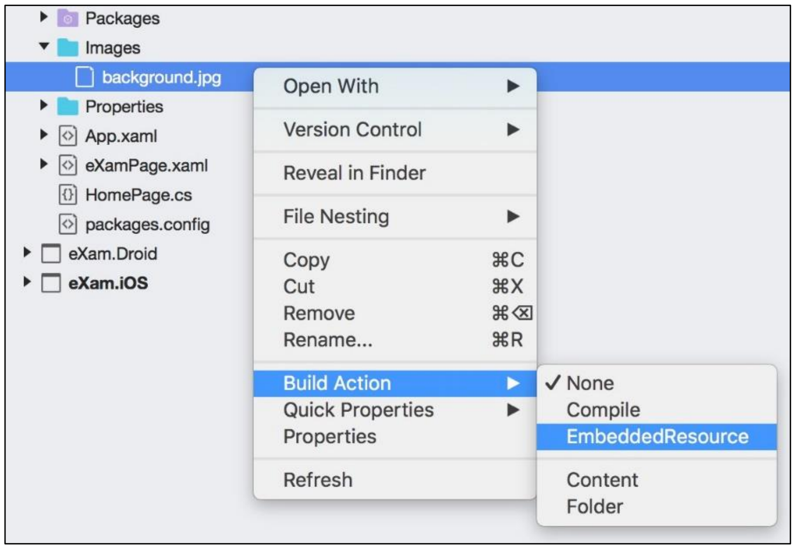


1. Create an **Images** folder in the PCL project.
    a. Locate **background.jpg** in the provided Assets folder and add it to your **Images** folder.
    b. Set the build action for **background.jpg** to **EmbeddedResource**.



background.jpg File Properties

Build Action	Embedded Resource
Copy to Output Directory	Do not copy
Custom Tool	



2. Open **HomePage.cs** and locate the constructor.

   a. Create an **Image** object. Set its **Source** using the static **ImageSource.FromResource** method passing the string **"eXam.Images.background.jpg"** as the resource. Set its **Aspect** to **AspectFill.** This string is a combination of your PCL's default namespace (*eXam*), the name of the folder (*Images*), and the name of the image file (*background.jpg*). This string is case-sensitive.
   b. Above the code where you added the button to the layout, add the background image to the **Children** collection of the **AbsoluteLayout**. Use (0, 0, 1, 1) as the Rectangle parameter and use **SizeProportional** for the **AbsoluteLayoutFlags**. The order you add the children determines their z-order; you should add the image first so it appears below the button.

```
this.Content = layout;

Image image = new Image
{
    Source = ImageSource.FromResource("eXam.Images.background.jpg"),
    Aspect = Aspect.AspectFill
};

layout.Children.Add(
    view: image,
    bounds: new Rectangle(x: 0, y: 0, width: 1, height: 1),
    flags: AbsoluteLayoutFlags.SizeProportional);

Button button = new Button
{
    Text = "Start eXam!",
    BackgroundColor = Color.LightCoral,
    Font = Font.SystemFontOfSize(20)
};

layout.Children.Add(
    view: button,
    bounds: new Rectangle(x: 0.5, y: 0.5, width: 150, height: 60),
    flags: AbsoluteLayoutFlags.PositionProportional);
```

3. Run the application to test your work. You should see the Start button over the new background image.

4 Set up stack navigation

In this part, you will add a **NavigationPage** that provides the infrastructure needed to switch between app pages. In a later part, you will use the **NavigationPage** to change pages in response to button taps.

1. Open the Application class: **App.xaml.cs**.
    a. In the constructor, instantiate a new **NavigationPage**, passing an instance of **HomePage** into the constructor.
    b. Assign the **NavigationPage** to the app's **MainPage**. This will replace the previous assignment of a **HomePage** object.

```
public App()
{
    InitializeComponent();

    NavigationPage navigationPage = new NavigationPage(new HomePage());
    MainPage = navigationPage;
}
```

2. Run the application; notice you now have a navigation bar showing on the landing page. You'll use the navigation bar on other pages but you don't need it on the landing page.

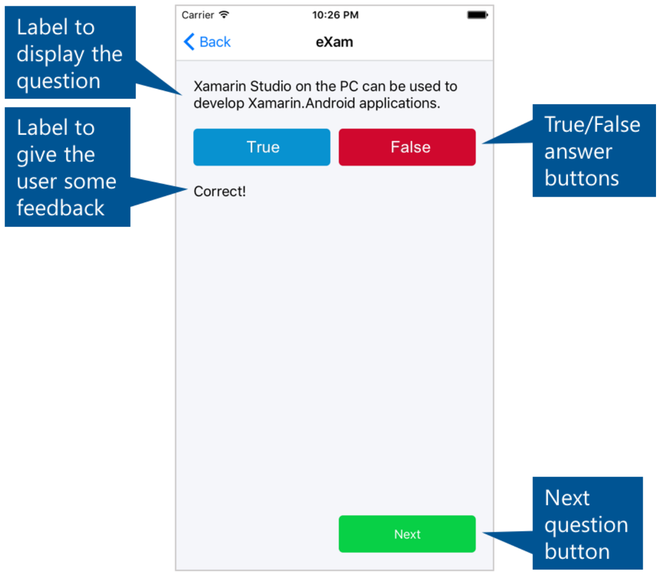
3. Open **HomePage.cs** and locate the constructor.
    a. Hide the navigation bar using the static **NavigationPage.SetHasNavigationBar** method.

```
public HomePage()
```

```
{
    NavigationPage.SetHasNavigationBar(page: this, value: false);

    // ...
}
```

5 Add a XAML page

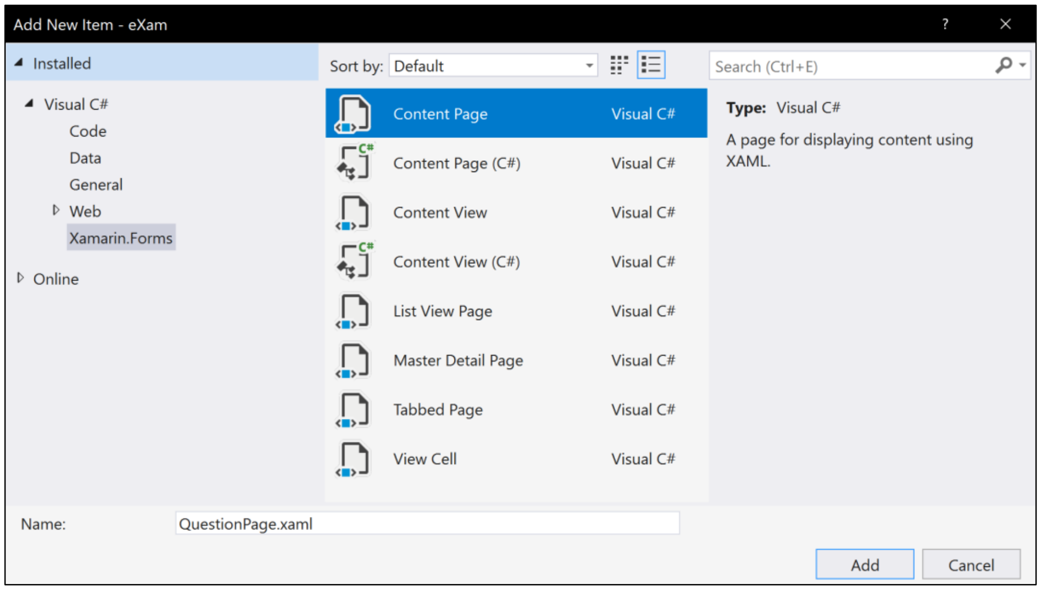
In this part, you will add a page to display a quiz question (see below). This part is UI-only so the buttons will not have any behavior yet and the labels will display place-holder text.


Label to display the question
Label to give the user some feedback
True/False answer buttons
Next question button
Carrier
10:26 PM
Back
eXam
True
False
Next

Xamarin Studio on the PC can be used to develop Xamarin.Android applications.

Correct!

1. Add a new Content Page (XAML) page to the PCL project. Name it **QuestionPage**.



Name: QuestionPage.xaml

2. Open **QuestionPage.xaml**.
   a. Remove the StackLayout and add a **Grid** with 4 rows and 2 columns to the **Content** of the **Page**.
      i. The height of rows 0, 1, and 3 should be **Auto** sized to match their largest child.
      ii. The height of row 2 should be set to "*****" so it takes up all the remaining vertical space.
      iii. The columns should be equal width, which is the default.

   To get you started, below is sample showing how to define the first row in XAML, the other rows and the columns are similar:

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto" />
    ...
</Grid.RowDefinitions>
```

3. Add 3 buttons and 2 labels to the Grid as shown in the screenshot above.
   a. Add a label for the question text to Row=0, Column=0, with a ColumnSpan=2
   b. Add a button for True to Row=1, Column=0
   c. Add a button for False to Row=1, Column=1
   d. Add a label for the answer text to Row=2, Column=0, ColumnSpan=2
   e. Add a button for Next to Row=3, Column=1

4. Provide default **Text** for the labels so you can see them during testing.

```
<Grid Padding="10">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Label Text="Question Text Default" Grid.Row="0" Grid.Column="0"
Grid.ColumnSpan="2" />
    <Button Text="True" Grid.Row="1" Grid.Column="0" />
    <Button Text="False" Grid.Row="1" Grid.Column="1" />
    <Label Text="Answer Text Default" Grid.Row="2" Grid.Column="0"
Grid.ColumnSpan="2" />
    <Button Text="Next" Grid.Row="3" Grid.Column="1" />
</Grid>
```

5. Set the Page's **Title** property to "eXam". This can be done in either the .xaml or .xaml.cs file.

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

6 Turn on XAMLC

In this part, you will instruct the build system to compile your XAML pages. This gives you more compile-time validation and improves runtime performance.

1. Open the Application class: **App.xaml.cs**.

2. Add the **XamlCompilationAttribute** to the top of the file to enable assembly-wide XAML compilation. Be sure to add the attribute outside of any namespace declarations, and include the **assembly:** prefix.

```
using Xamarin.Forms;
[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
```

```
namespace eXam
{
    public partial class App : Application
    {
       // . . .
    }
}
```

7 Navigate on button tap

In this part, you will add behavior to the landing-page button so it navigates to the question page.

1. Open **HomePage.cs**.
   a. Locate the code in the constructor where you create the button.
   b. Subscribe to the button's **Clicked** event using any subscription technique you prefer (named method, lambda, delegate).
   c. In your handler, navigate the app to **QuestionPage**. Use the page's inherited **Navigation** property and its **PushAsync** method to move the app to the new page.

```
button.Clicked += async (sender, args) =>
{
    await this.Navigation.PushAsync(new QuestionPage());
};
```

2. Run the application and tap the Start button.

3. This is the first time you will be able to see the **QuestionPage** UI running live. Spend a few minutes revising the UI layout if needed. Hint: Add Padding of 10 to the GridLayout.

8 Load game data

In this part, you will add pre-written classes for the game logic and an XML file containing the questions. You'll hold a static instance of the game class in the Application class so it's easy to access from all parts of the app. You will need to add several "using" statements during this lab.

1. Locate **QuizQuestion.cs**, **QuizQuestionXmlSerializer.cs**, and **Game.cs** in the provided Assets folder and add them to the PCL project.

2. Open **QuizQuestion.cs**. This is the model object for a question. It is provided to save you some typing.

3. Open **QuizQuestionXmlSerializer.cs**. This provides methods to translate **QuizQuestion** objects to/from XML. Again, this is fairly straightforward C#/.NET code and is provided to save you time.

4. Open **Game.cs**. This implements the game logic to track user progress, remember user answers, and calculate their score. Notice how this too is pure C#/.NET – it is platform-independent code that can be shared across all the platforms you want to support: iOS, Android, Windows, MacOS, etc.

5. Add a new folder named **Data** to the PCL project.
   a. Locate **questions.xml** in the Assets folder. Add it to the **Data** folder.
   b. Set the **Build Action** of **questions.xml** to **Embedded Resource**.

6. Open the Application class: **App.xaml.cs**, and locate the **OnStart** method. You will be adding code to this method.
   a. Files packaged as Embedded Resource are available through the Assembly object for that assembly. Use **typeof(App).GetTypeInfo().Assembly** to get a reference to the assembly object containing the App class and store it in a local variable.
   b. Use the **GetManifestResourceStream** method from the **Assembly** class to open the questions.xml file. The filename needs to include the assembly name and the folder name: **eXam.Data.questions.xml**.
   c. You now have a **Stream** object. Use a System.IO.**StreamReader** and its **ReadToEnd** method to convert the file contents into a string.
   d. Use the provided **QuizQuestionXmlSerializer** to deserialize the stream containing the XML into a **List<QuizQuestion>**.
   e. In the **App** class, create a public static property, of type **Game**, named **CurrentGame**.
   f. Instantiate a **Game** object, passing in the list of quiz questions; store the game object in the **CurrentGame** property.

```
public static Game CurrentGame { get; set; }

protected override async void OnStart()
{
    // Handle when your app starts
    var assembly = typeof(App).GetTypeInfo().Assembly;
    var resourceStream = assembly.GetManifestResourceStream("eXam.Data.questions.xml");
    var reader = new StreamReader(resourceStream);
    var xml = reader.ReadToEnd();
    var embeddedQuestions = QuizQuestionXmlSerializer.Deserialize(xml);
    CurrentGame = new Game(embeddedQuestions);
}
```

9 Cache game data

In this part, you will add code to cache and load the quiz questions from the on-device file system. This is preparation for a later step when you will download the questions from an Azure Web Service – the idea is to always have a cached copy of the latest questions on-device so the user can still play the game even if the app does not have internet connectivity.

Working with the on-device file system requires specific code for each platform. You will do this with an interface in the PCL that is implemented by a separate class in each of the platform-specific projects.

1. Add a new interface named **IFileHelper.cs** to the PCL.

2. Add the code below to the **IFileHelper.cs** file. Be sure to make the interface public. This interface is intentionally kept simple; for example, notice how **LoadLocalFileAsync** and **SaveLocalFileAsync** transfer the file contents as a single string. If you can think of a more sophisticated design; for example, you could use **Stream**s instead of strings, please feel free to experiment as you have time.

```
public interface IFileHelper
{
  Task<string> LoadLocalFileAsync (string filename);
  Task<bool>   SaveLocalFileAsync (string filename, string data);

  string GetNameWithPath (string filename);
}
```

3. In the assets folder, you will find **PlatformFileHelper.cs** containing an implementation of the interface for each platform. Add the appropriate file to each platform-specific project.

4. Use an assembly-level attribute to register **PlatformFileHelper** with the **DependencyService** in each platform-specific project. For simplicity, this can be done in each **PlatformFileHelper.cs** file.

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

namespace eXam.Droid
{
    public class PlatformFileHelper : IFileHelper
    {
        // . . .
    }
}
```

5. Open the Application class: **App.xaml.cs**, and locate the **OnStart** method. You will be adding code to this method.
   a) Use the **DependencyService** to get an **IFileHelper** implementation. The **DependencyService** will give you the correct implementation based on the current runtime platform.
   b) Use the **FileHelper** to load a file named **cachedquestions.xml** using the **LoadLocalFileAsync** method (the xml file doesn't exist yet but we'll create it shortly). Notice this method returns a **Task<string>** so you'll need to add the **async** keyword to the **OnStart** method and await this call.
   c) Test if the load fails by checking if the returned string is **null**. If it failed, you don't have a cached copy on disk yet, so run the existing code that loads the XML from the embedded file in the PCL. Notice how we are setting up a fallback: use the cache if it is available, otherwise use the embedded file. Later, we will extend this concept to include a question file downloaded from an Azure Web Service.
   d) After loading from the PCL, use the **FileHelper** to save the embedded XML into a file named **cachedquestions.xml**. The next time the app runs, this cached version will be available and the app won't need to load from the embedded file.

6. At this point, you will have a string that represents the XML. Your existing code that converts the string to a **List<QuizQuestion>** and then uses the **List<QuizQuestion>** to instantiate a **Game** object will remain the same.

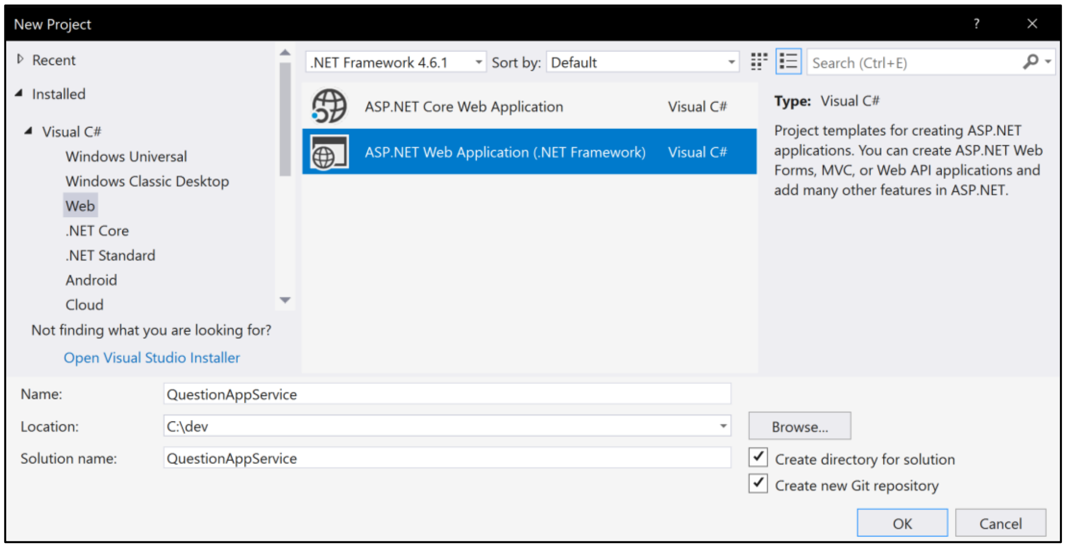
```
protected override async void OnStart()
{
    // . . .

    IFileHelper fileHelper = DependencyService.Get<IFileHelper>();
    var cachedQuestions = await fileHelper.LoadLocalFileAsync("cachedquestions.xml");
    if (string.IsNullOrWhiteSpace(cachedQuestions))
    {
        await fileHelper.SaveLocalFileAsync("cachedquestions.xml", xml);
    }
}
```

10 Create the Azure Mobile App and Publish it

In this part, you will create our Azure Mobile App and publish it. You will use Visual Studio to create a project and to Publish the Azure Mobile App.

1. Open Visual Studio and create a new solution.

2. Select **ASP.NET Web Application (.NET Framework)** from the Web category and give it the name "QuestionAppService".


New Project

Recent
Installed
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Android
Cloud
Not finding what you are looking for?
Open Visual Studio Installer
.NET Framework 4.6.1
Sort by: Default
Search (Ctrl+E)
ASP.NET Core Web Application Visual C#
ASP.NET Web Application (.NET Framework) Visual C#

Type: Visual C#

Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

Name: QuestionAppService

Location: C:\dev

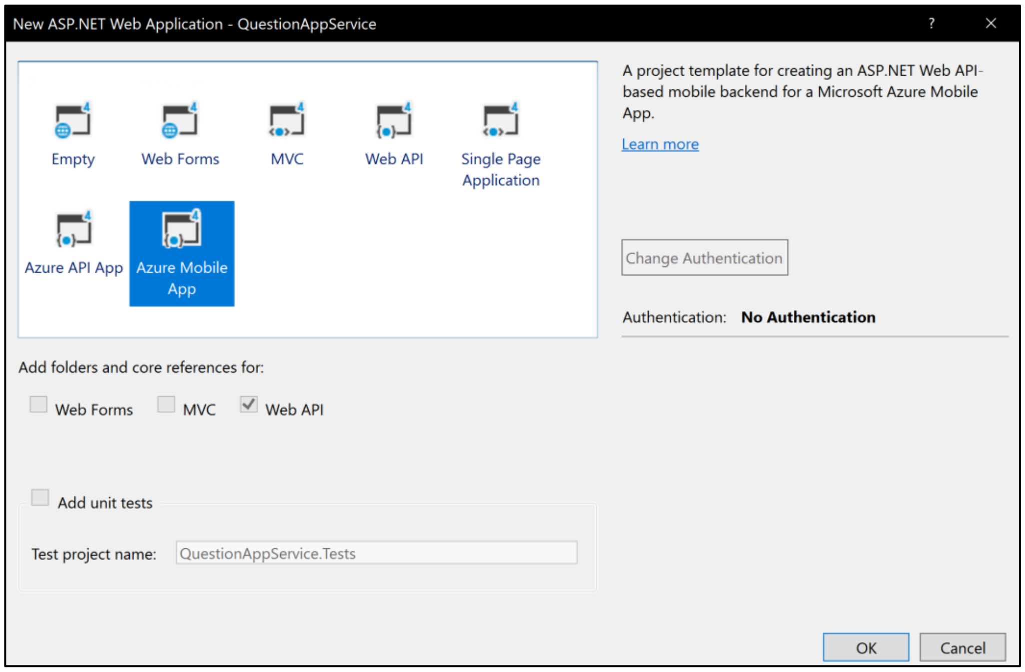
Solution name: QuestionAppService

☑ Create directory for solution

☑ Create new Git repository

3. Uncheck the "Add Application Insights" option if it's checked in the dialog

4. On the web application screen, select the Azure Mobile App template. Click **OK**


New ASP.NET Web Application - QuestionAppService

A project template for creating an ASP.NET Web API-based mobile backend for a Microsoft Azure Mobile App.

Authentication: **No Authentication**

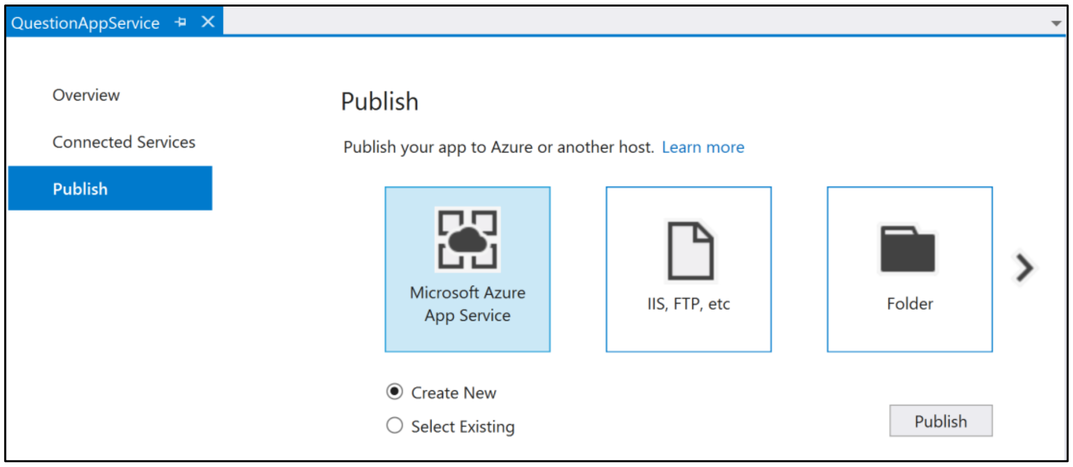
Add folders and core references for:

☐ Web Forms ☐ MVC ☑ Web API

☐ Add unit tests

Test project name: QuestionAppService.Tests

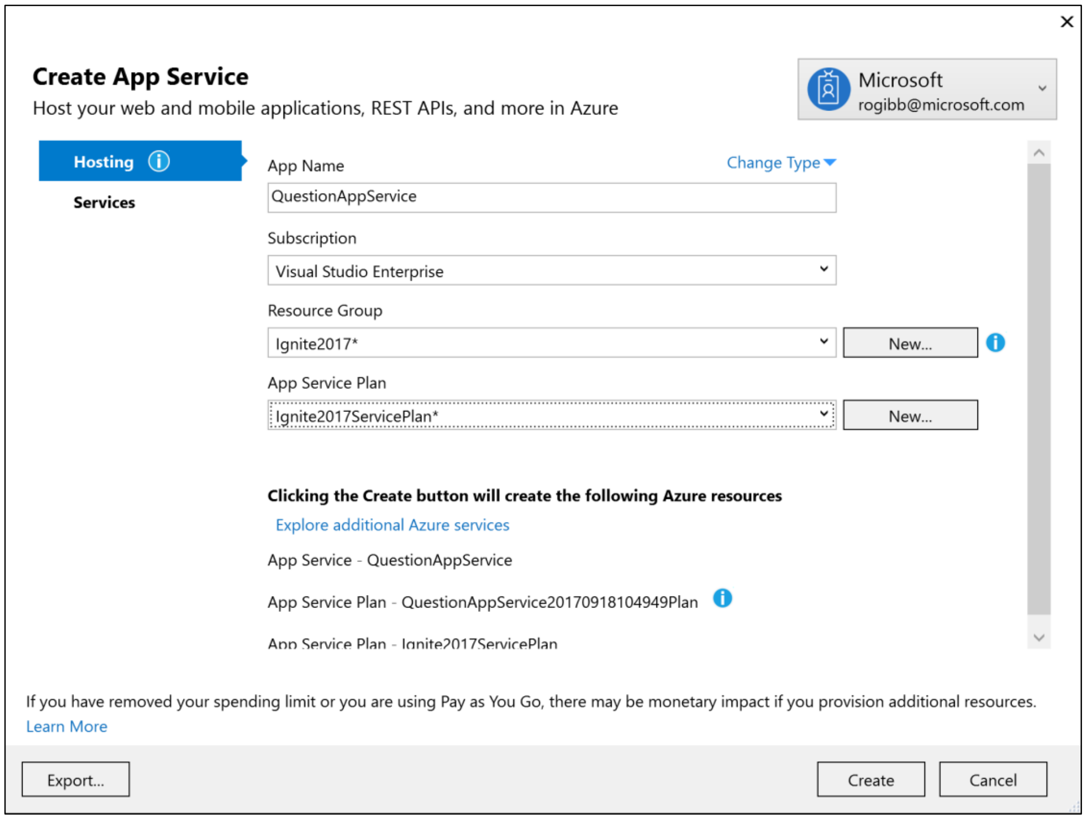
5. Once your web application is created, an Overview screen will display. Switch to the **Publish** tab to start setting up your Azure deployment configuration. Make sure **Microsoft Azure App Service** and the **Create New** option is selected and click **Publish** to create the application.



Publish your app to Azure or another host. Learn more

[✓ Create New]
Select Existing

6. This will open the publishing configuration dialog. We want to host this in Azure.



Create App Service

Host your web and mobile applications, REST APIs, and more in Azure

Microsoft
rogibb@microsoft.com

App Name: QuestionAppService

Subscription: Visual Studio Enterprise

Resource Group: Ignite2017*

App Service Plan: Ignite2017ServicePlan*

Clicking the Create button will create the following Azure resources

Explore additional Azure services

App Service - QuestionAppService

App Service Plan - QuestionAppService20170918104949Plan

App Service Plan - Ignite2017ServicePlan

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources. Learn More

7. When you have filled in the required data, click **Create** to add your service definition to Azure. You can also request to pin this new app to your dashboard if you like.

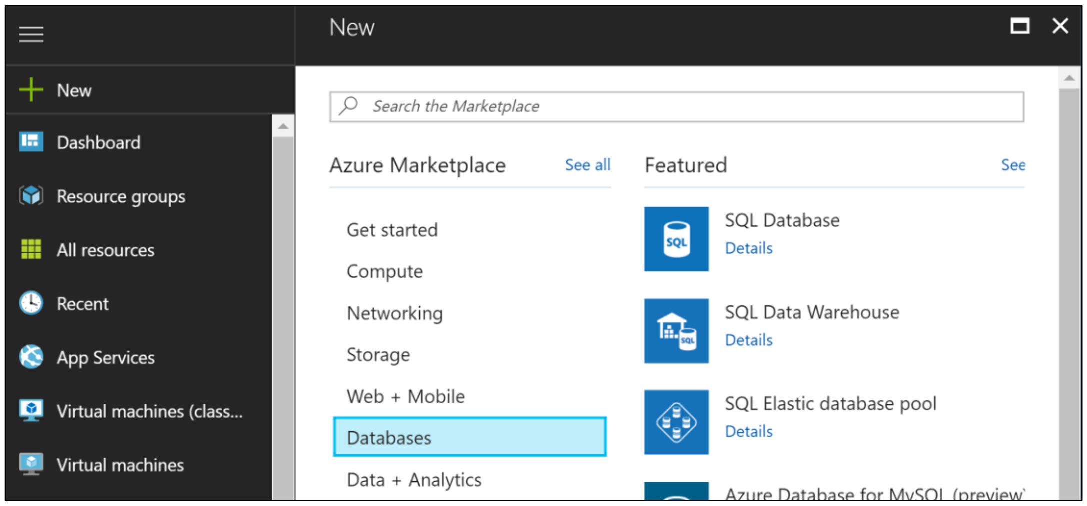
8. Depending on your version of Visual Studio, it may automatically publish to Azure. If not, go ahead and **Build** the project, then use the **Build > Publish** option to publish your server code to Azure. This option is also available from the project's context menu in the Solution Explorer.

11a Add SQL Database

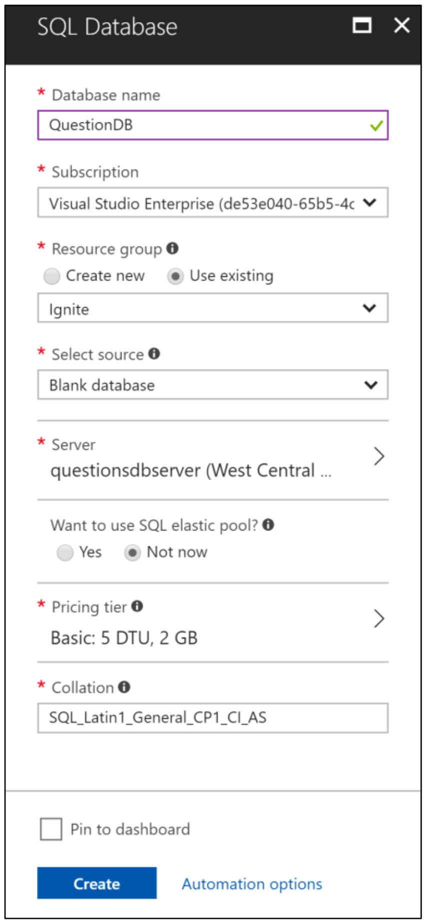
In this part, you will add a SQL database too your Azure Mobile App.

1. Open a browser window and login to the Azure management portal with your account.

2. You can either use the **+ New > Databases > SQL Database** option, or click the **SQL databases** option in the sidebar and then click the **New** toolbar button



3. Fill out the required database configuration:


SQL Database

* Database name
QuestionDB

* Subscription
Visual Studio Enterprise (de53e040-65b5-4c

* Resource group
Create new [✓ Use existing]
Ignite

* Select source
Blank database

* Server
questionsdbserver (West Central ...

Want to use SQL elastic pool?
Yes [✓ Not now]

* Pricing tier
Basic: 5 DTU, 2 GB

* Collation
SQL_Latin1_General_CP1_CI_AS

Pin to dashboard

4. Click **Create** to create the database. This will take a few minutes - particularly if you had to define a new server to put the database on.

Now that we have a database defined, let's allow our app service to access it by defining a connection string.

1. Open your app service - you can either go back to the Azure dashboard by clicking on "Microsoft Azure" in the header if you pinned your service, or click the "App Services" link in the sidebar to list all your defined app services. Locate your app and click on it.

2. Scroll down to the "Mobile" section in your app properties blade and select the "Data Connections" item.

3. Click the **+ Add** toolbar button to add a new connection.

4. Select "SQL Database" from the **Type** dropdown and then click the "SQL Database" entry to configure the database.

5. This will open a new blade to select or create a database and your database you just created should show up in the list - select it and click .

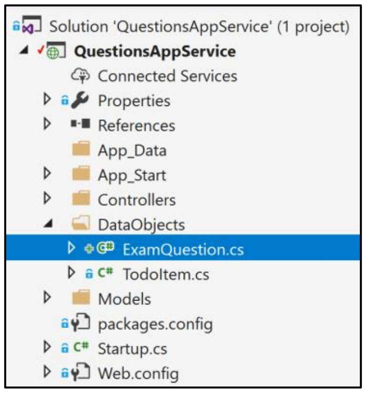
6. Select the "Connection String" item from the blade - this will open a sub-blade where you can set the user/password for the database; make sure to use the same credentials you setup when you created the database.

7. The "Name" will pre-populate with "MS_TableConnectionString" which is the value we want. Click **OK** to create the connection string.

11b Add a DTO, Controller, and Azure Table

In this part, you will add a DTO for our server side model object, a controller for that DTO, and create an Azure Table with data.

1. In your App Service project code in Visual Studio, create a new class file named "ExamQuestion" in the **Data Objects** folder.



2. Have the class derive from Microsoft.Azure.Mobile.Server.EntityData.

```
using Microsoft.Azure.Mobile.Server;

namespace QuestionsAppService.DataObjects
{
    public class ExamQuestion : EntityData
    {
    }
}
```

3. Add the following public properties to the class:
    a. string Id
    b. string Question
    c. bool Answer
    d. string Explanation

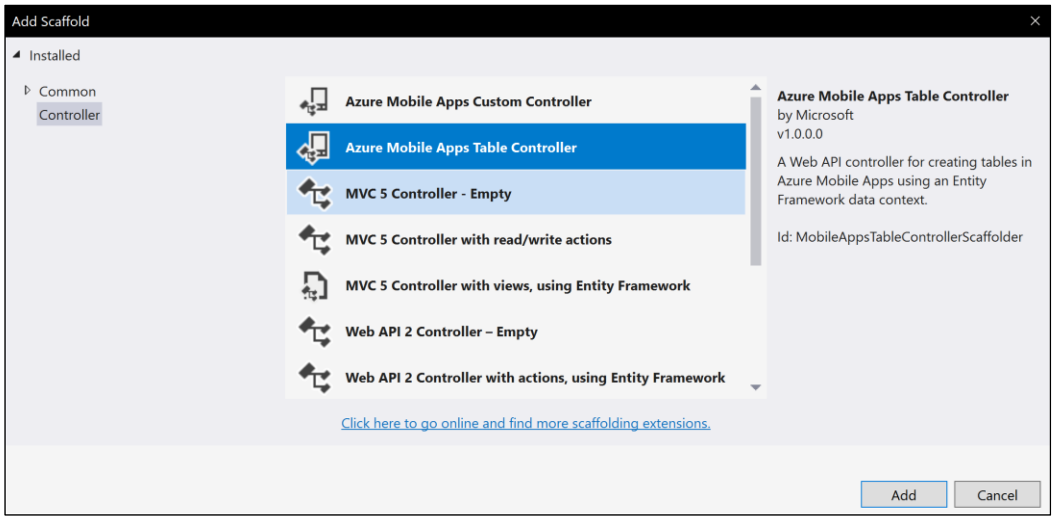
```
public class ExamQuestion : EntityData
{
    public string Id { get; set; }
    public string Question { get; set; }
    public bool Answer { get; set; }
    public string Explanation { get; set; }
}
```

4. Add a **[Table("questions")]** attribute to your class definition. This attribute is in the System.ComponentModel.DataAnnotations.Schema namespace.

```
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.Azure.Mobile.Server;

namespace QuestionsAppService.DataObjects
{
    [Table("questions")]
    public class ExamQuestion : EntityData
    {
    }
}
```

5. Right-click on the Controllers folder and select Add > Controller.

6. Select the **Azure Mobile Apps Table Controller** from the wizard.


Add Scaffold

Installed
- Common
- Controller

- Azure Mobile Apps Custom Controller
- Azure Mobile Apps Table Controller
- MVC 5 Controller - Empty
- MVC 5 Controller with read/write actions
- MVC 5 Controller with views, using Entity Framework
- Web API 2 Controller – Empty
- Web API 2 Controller with actions, using Entity Framework

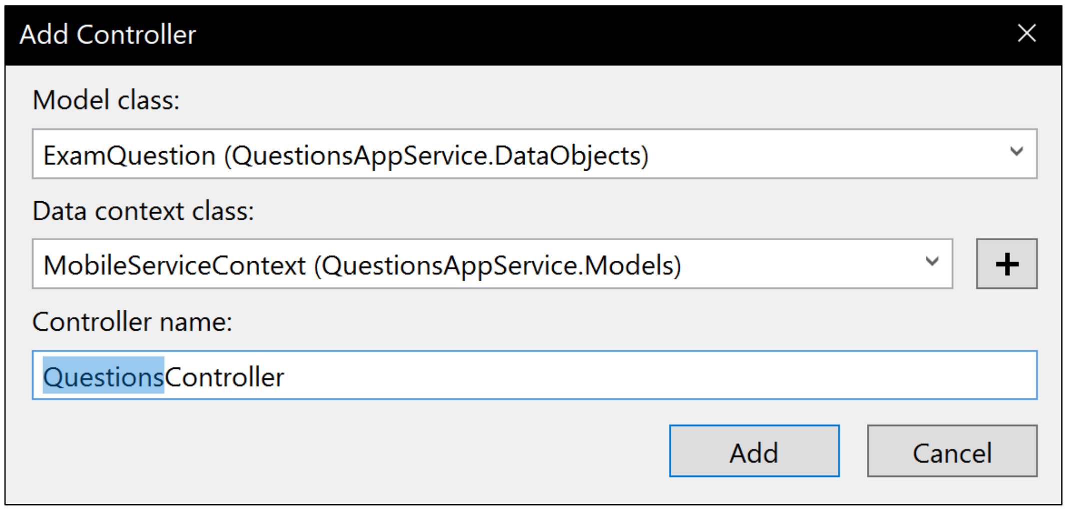
Azure Mobile Apps Table Controller
by Microsoft
v1.0.0.0

A Web API controller for creating tables in Azure Mobile Apps using an Entity Framework data context.

Id: MobileAppsTableControllerScaffolder

Click here to go online and find more scaffolding extensions.

7. In the **Add Controller** dialog:
   a. Select your ExamQuestion DTO from the Model class dropdown.
   b. Select the MobileServiceContext from the Data context class dropdown.
   c. Name your new controller "**QuestionsController**". This will set the endpoint to "questions"

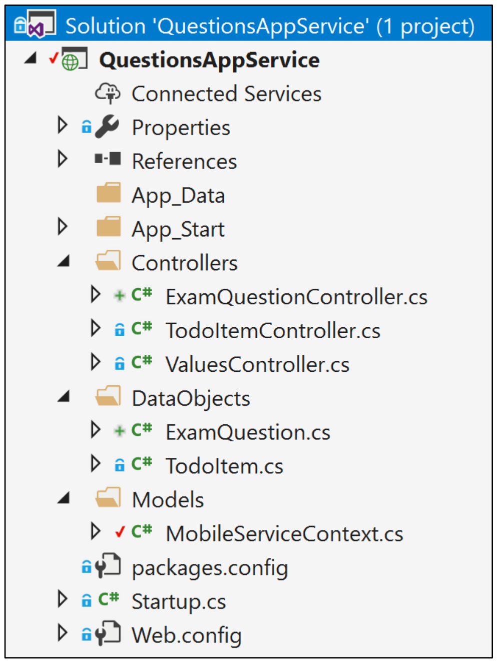

Add Controller

Model class: ExamQuestion (QuestionsAppService.DataObjects)

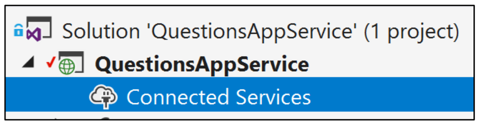
Data context class: MobileServiceContext (QuestionsAppService.Models)

Controller name: QuestionsController

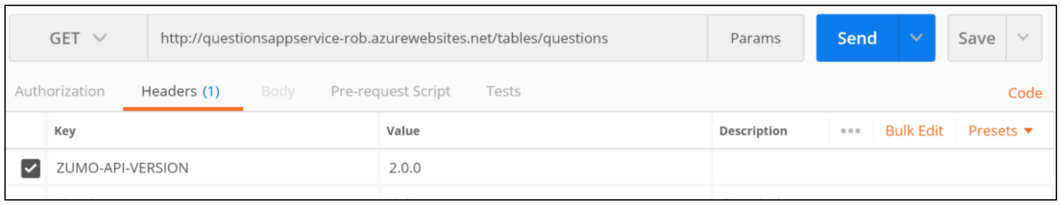
8. Click **Add** to add the controller code. This will add a new class into the **Controllers** folder and also add a new DbSet<ExamQuestion> property into the MobileServiceContext class contained in the **Models** folder.



9. Double click on the **Connected Services** node in the project, and publish the site to Azure (or run it locally).



10. Check it with Postman (http://www.getpostman.com) or Insomnia (http://insomnia.rest) if you want.
    a. The **GET** URL will be http://<your-app-service>.azurewebsites.net/tables/questions
    b. Be sure to add a header named "**ZUMO-API-VERSION**" with a value of "2.0.0"
    c. The return value will be an empty array


GET http://questionsappservice-rob.azurewebsites.net/tables/questions

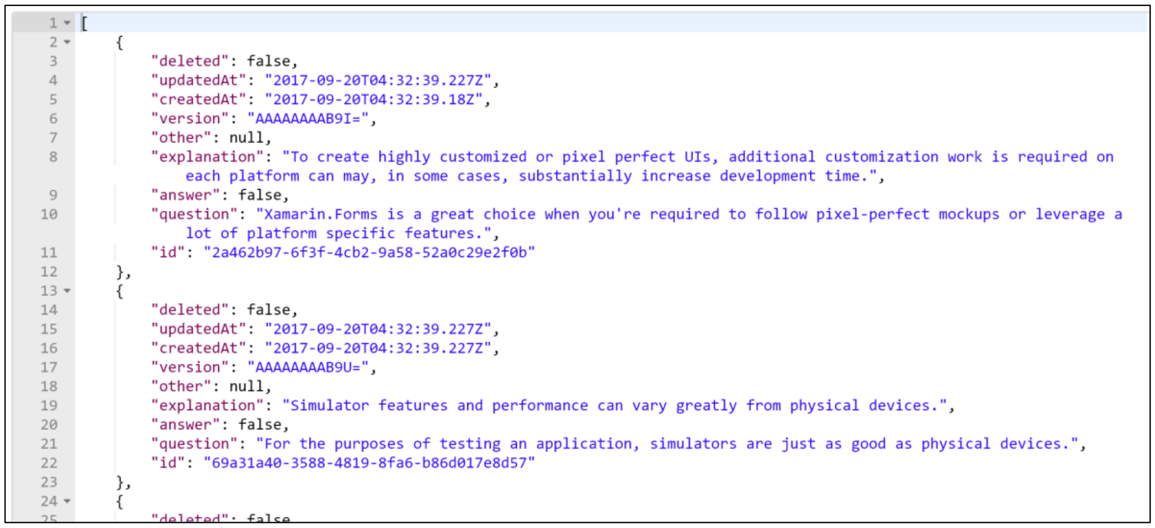
Key	Value	Description
✓ ZUMO-API-VERSION	2.0.0	

11. To add the seed data to the table, add the **DataFactory** class found in the Assets folder to your project.

12. Open the **Startup.MobileApp.cs** file in the App_Start folder in the Solution Explorer.
    a. Locate the database initializer class named **MobileServiceInitializer**. The base class used here, **CreateDatabaseIfNotExists**, will create the tables the first time and throw an error if the schema is ever changed in the future. We don't want that behavior so change it to **DropCreateDatabaseIfModelChanges**.
    b. The initialization process will also call the defined Seed method to populate any tables with data. There might be some code here to initialize the TodoItem DTO which was added as part of the initial project template. You can remove that code and instead add all the questions from the DataFactory.Questions static list to the context.ExamQuestions database set using the AddRange method.

```
public class MobileServiceInitializer :
DropCreateDatabaseIfModelChanges<MobileServiceContext>
{
    protected override void Seed(MobileServiceContext context)
    {
        context.ExamQuestions.AddRange(DataFactory.Questions);
        base.Seed(context);
    }
}
```

13. Publish the site to Azure. Once the site is live check your new endpoint (<URL>/tables/examquestion) using your REST client



```
[
    {
        "deleted": false,
        "updatedAt": "2017-09-20T04:32:39.227Z",
        "createdAt": "2017-09-20T04:32:39.18Z",
        "version": "AAAAAAAAB9I=",
        "other": null,
        "explanation": "To create highly customized or pixel perfect UIs, additional customization work is required on
            each platform can may, in some cases, substantially increase development time.",
        "answer": false,
        "question": "Xamarin.Forms is a great choice when you're required to follow pixel-perfect mockups or leverage a
            lot of platform specific features.",
        "id": "2a462b97-6f3f-4cb2-9a58-52a0c29e2f0b"
    },
    {
        "deleted": false,
        "updatedAt": "2017-09-20T04:32:39.227Z",
        "createdAt": "2017-09-20T04:32:39.227Z",
        "version": "AAAAAAAAB9U=",
        "other": null,
        "explanation": "Simulator features and performance can vary greatly from physical devices.",
        "answer": false,
        "question": "For the purposes of testing an application, simulators are just as good as physical devices.",
        "id": "69a31a40-3588-4819-8fa6-b86d017e8d57"
    },
    {
        "deleted": false
```

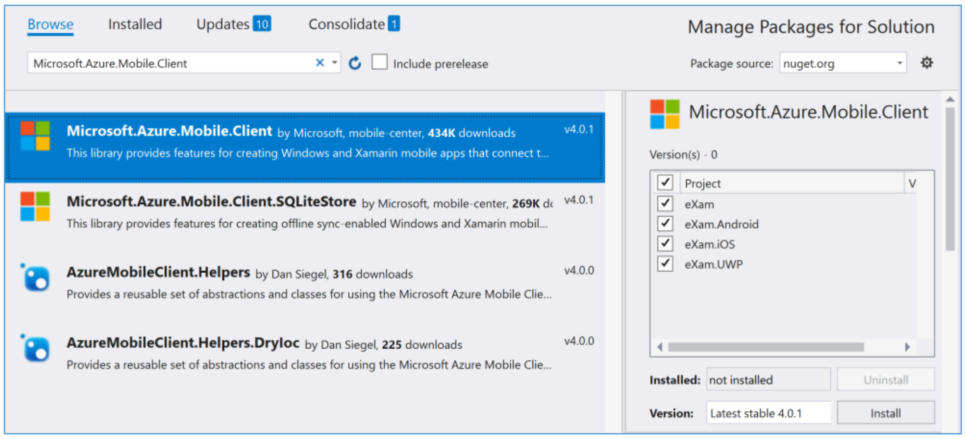
12a Add Azure Support to Our Client Application

In this part, we will add the Azure client SDK as well as a Connectivity plugin to make sure we are online and Newtonsoft.Json Nuget package to deserialize the data we receive.

1. Open the **eXam** mobile client solution in Visual Studio.

2. Add nuget packages to the **eXam (Portable)**, **eXam.Android**, **eXam.iOS**, and **eXam.UWP** projects.
    a. Add the **Microsoft.Azure.Mobile.Client** package
    b. Add the **Xam.Plugin.Connectivity** package
    c. Add the Newtonsoft **Json.NET** package
    *d. Be sure to add these Nuget packages to **ALL** four projects*

3. Add the **Microsoft.Net.Http** package to the PCL project



Browse
Installed
Updates 10
Consolidate 1
Manage Packages for Solution
Microsoft.Azure.Mobile.Client
Include prerelease
Package source: nuget.org

Microsoft.Azure.Mobile.Client by Microsoft, mobile-center, **434K** downloads v4.0.1
This library provides features for creating Windows and Xamarin mobile apps that connect t...

Microsoft.Azure.Mobile.Client.SQLiteStore by Microsoft, mobile-center, **269K** dc v4.0.1
This library provides features for creating offline sync-enabled Windows and Xamarin mobil...

AzureMobileClient.Helpers by Dan Siegel, **316** downloads v4.0.0
Provides a reusable set of abstractions and classes for using the Microsoft Azure Mobile Clie...

AzureMobileClient.Helpers.DryIoc by Dan Siegel, **225** downloads v4.0.0
Provides a reusable set of abstractions and classes for using the Microsoft Azure Mobile Clie...

Microsoft.Azure.Mobile.Client

Version(s) - 0

Project	V
✓ eXam	
✓ eXam.Android	
✓ eXam.iOS	
✓ eXam.UWP	

Installed: not installed — Uninstall

Version: Latest stable 4.0.1 — Install

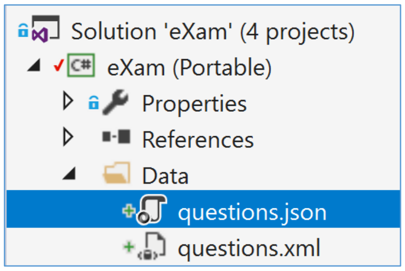
4. Open the **MainActivity.cs** file in the Xamarin.Android project. Since this is a Xamarin.Forms app, the main Activity only launches once per app-launch and we can do our initialization here.
    a. Add a call to **Microsoft.WindowsAzure.MobileServices.CurrentPlatform.Init();** in the **OnCreate** override - you can place it just before the **Forms.Init** call.

```
protected override void OnCreate(Bundle bundle)
{
    TabLayoutResource = Resource.Layout.Tabbar;
    ToolbarResource = Resource.Layout.Toolbar;

    base.OnCreate(bundle);

    Microsoft.WindowsAzure.MobileServices.CurrentPlatform.Init();

    global::Xamarin.Forms.Forms.Init(this, bundle);
    LoadApplication(new App());
}
```

5. Open the **AppDelegate.cs** file in the Xamarin.iOS project.
    a. Add a call to **Microsoft.WindowsAzure.MobileServices.CurrentPlatform.Init();** in the **FinishedLaunching** override - you can place it just before the **Forms.Init** call.

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
{
    Microsoft.WindowsAzure.MobileServices.CurrentPlatform.Init();

    global::Xamarin.Forms.Forms.Init();
    LoadApplication(new App());

    return base.FinishedLaunching(app, options);
}
```

6. In the **eXam** PCL project
    a. Locate questions.json in the Assets folder. Add it to the Data folder.
    b. Set the Build Action of questions.json to Embedded Resource.



7. In the **eXam** PCL, open the Application class: **App.xaml.cs** and locate the **OnStart** method.
    a. Use the static **CrossConnectivity.Current.IsConnected** property to check if the application is connected to the internet.
    b. In the **OnStart** method, change the name of the loaded question file from questions.xml to questions.json.
    c. Change the name of the cache file from cachedquestions.xml to cachedquestions.json.
    d. Change the deserialize code so it uses **JsonConvert.DeserializeObject<List<QuizQuestion>>()** from the Json.NET package instead of **QuizQuestionXmlSerializer.Deserialize**.
    (continued on next page)

```
protected override async void OnStart()
{
    // Handle when your app starts
    var assembly = typeof(App).GetTypeInfo().Assembly;
    var resourceStream =
assembly.GetManifestResourceStream("eXam.Data.questions.json");
    var reader = new StreamReader(resourceStream);
    var json = reader.ReadToEnd();
    var embeddedQuestions =
JsonConvert.DeserializeObject<List<QuizQuestion>>(json);
    CurrentGame = new Game(embeddedQuestions);

    IFileHelper fileHelper = DependencyService.Get<IFileHelper>();
    string cachedQuestions = string.Empty;

    if (CrossConnectivity.Current.IsConnected)
    {
        // We're online
    }
    else
    {
        // We're offline.  Load the questions from the local cache
        cachedQuestions = await
fileHelper.LoadLocalFileAsync("cachedquestions.json");
    }

    if (string.IsNullOrWhiteSpace(cachedQuestions))
    {
        await fileHelper.SaveLocalFileAsync("cachedquestions.json", json);
    }
}
```

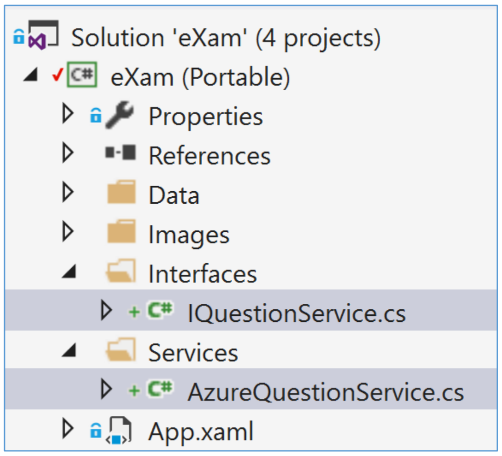
8. If you do not have connectivity, execute the previous code that loads the questions locally. In either case, you should still cache the JSON string in a local file.

12b Client Side DTO and Service

In this part, we will set up our client side DTO and retrieve the questions from Azure.

1. In the **eXam** PCL project, add a folder named **Interfaces**
    a. Add the existing **IQuestionService.cs** file from the Assets folder to the Interfaces folder. This is the interface we need to implement.

2. In the **eXam** PCL project, add a folder named **Services**
    a. Add a new class file named **AzureQuestionService.cs** to the Services folder.



```
Solution 'eXam' (4 projects)
  eXam (Portable)
    Properties
    References
    Data
    Images
    Interfaces
      IQuestionService.cs
    Services
      AzureQuestionService.cs
    App.xaml
```

3. Have the **AzureQuestionService** class implement **IQuestionService** - you can use the built-in IDE support to add each required method stub. Just leave the code to throw a **NotImplementedException** in place for now.
    a. Add a new private field of type **MobileServiceClient**.
    b. Add a new private method named **Initialize**, create a mobile service client object and assign it to your field. You will need the URL of the Azure service to pass into the constructor.
    c. Add a check into the Initialize method to see if the **MobileServiceClient** has been created (non-null) and if so, return. We want to be able to call this method multiple times, but only have the logic get executed once.
    d. Next, add a call to the Initialize method into each of your implementation methods - we will create it the first time we actually use the object (vs. when it is created).
    (continued on next page)

```
public class AzureQuestionService : IQuestionService
{
    private MobileServiceClient client;

    private void Initialize()
    {
        if (client == null)
        {
            client = new MobileServiceClient("http://<YOUR-APP_SERVICE-
URL>.azurewebsites.net");
        }
    }

    public Task<IEnumerable<QuizQuestion>> GetQuestionsAsync()
    {
        Initialize();
        throw new NotImplementedException();
    }
}
```

4. Open the **QuizQuestion.cs** source file.
    a. Add a **[Newtonsoft.Json.JsonObject(Title="questions")]** attribute to the class to fix the endpoint name.
    b. Add an Id property to match our service DTO.

```
namespace eXam
{
    [Newtonsoft.Json.JsonObject(Title = "questions")]
    public class QuizQuestion
    {
       public string Id          { get; set; }
       public string Question    { get; set; }
       public bool   Answer      { get; set; }
       public string Explanation { get; set; }
    }
}
```

5. Open the **AzureQuestionService.cs** file
    a. In the **Initialize** method, make a call to the **MobileServiceClient** instance's **GetTable** method using **QuizQuestion** as the DTO. This will return a **IMobileServiceTable<QuizQuestion>** which you should store in a class field **questionsTable**.
    b. In the **GetQuestionsAsync** method, use the table interface's **ReadAsync** method to read all the questions. This just returns a **Task<IEnumerable<ExamQuestion>>** which matches nicely with one of the retrieval methods we discussed in the class.
    (continued on next page)

```
public class AzureQuestionService : IQuestionService
{
    private MobileServiceClient client;

    private IMobileServiceTable<QuizQuestion> questionsTable;
    private void Initialize()
    {
        if (client == null)
        {
            client = new MobileServiceClient("http://<YOUR-APP-SERVICE-
URL.azurewebsites.net");
        }

        questionsTable = client.GetTable<QuizQuestion>();
    }

    public Task<IEnumerable<QuizQuestion>> GetQuestionsAsync()
    {
        Initialize();

        return questionsTable.ReadAsync();
    }
}
```

6. Open the **App.xaml.cs** file, and locate the **OnStart** method
   a. If you are connected to the network
      i. Get an instance of your **AzureAppService**.
      ii. Call **GetQuestionsAsync** on that instance. It will return an **IEnumerable<QuizQuestion>**, we will need to serialize it to JSON to save it to our file.
      iii. Go ahead and serialize the results and save them in the cachedQuestions field.
   b. If you are not connected to the network
      i. Move the logic to load the JSON from offline cache to the “else” part of the connectivity check.
      ii. We should put the call to get a Stream into a using statement so that it is disposed when we are finished with it.
   c. Keep the code to load the CurrentGame and store the offline cache outside of the connectivity check and at the end of the method.
   (continued on next page)

```
protected override async void OnStart()
{
    IFileHelper fileHelper = DependencyService.Get<IFileHelper>();
    string cachedQuestions;

    if (CrossConnectivity.Current.IsConnected)
    {
        // We're online
        IQuestionService questionService = new AzureQuestionService();
        var questionsList = await questionService.GetQuestionsAsync();
        cachedQuestions = JsonConvert.SerializeObject(questionsList);
    }
    else
    {
        // We're offline.  Load the questions from the local cache
        cachedQuestions = await
fileHelper.LoadLocalFileAsync("cachedquestions.json");
        if (string.IsNullOrWhiteSpace(cachedQuestions))
        {
            var assembly = typeof(App).GetTypeInfo().Assembly;
            using (var resourceStream =
assembly.GetManifestResourceStream("eXam.Data.questions.json"))
            {
                using (var reader = new StreamReader(resourceStream))
                {
                    cachedQuestions = reader.ReadToEnd();
                }
            }
        }
    }

    var embeddedQuestions =
JsonConvert.DeserializeObject<List<QuizQuestion>>(cachedQuestions);
    CurrentGame = new Game(embeddedQuestions);
    await fileHelper.SaveLocalFileAsync("cachedquestions.json", cachedQuestions);
}
```

7. Set a breakpoint at the end of this method, run the application and verify that questions show up from Azure

13 Use data binding to display a question

In this part, you will use data binding to display a question on the question page.

1. Open **QuestionPage.xaml.cs**.
    a. Add a constructor that accepts a **QuizQuestion** and assign the **QuizQuestion** to the **BindingContext** of the page. You will also need to call **InitializeComponent** from your new constructor.

```
public QuestionPage(QuizQuestion quizQuestion)
{
    this.BindingContext = quizQuestion;
    InitializeComponent();
}
```

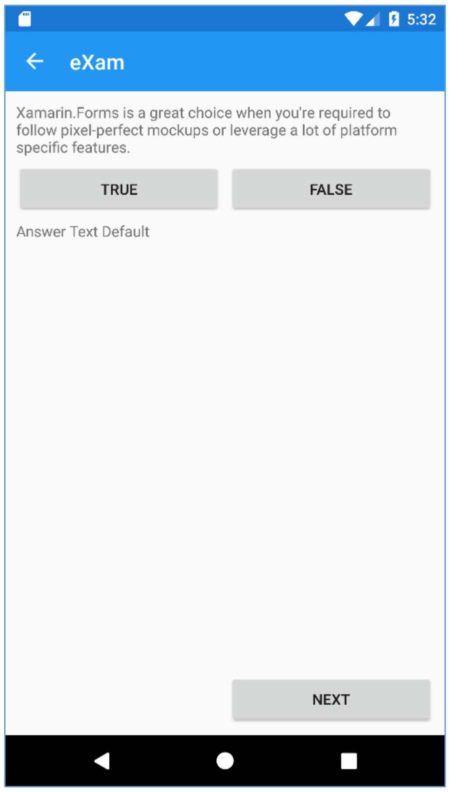
2. Open **HomePage.cs** and locate the click handler for the start button.
    a. During the navigation, pass in the **App.CurrentGame.CurrentQuestion** property to the **QuestionPage**'s constructor.

```
button.Clicked += async (sender, args) =>
{
    await this.Navigation.PushAsync(new
QuestionPage(App.CurrentGame.CurrentQuestion));
};
```

3. Open **QuestionPage.xaml** and locate the **Label** that displays the question text.
    a. Bind the label's **Text** property to the Binding Context's **Question** property.

```
<Label Text="{Binding Question}" Grid.Row="0" Grid.Column="0"
Grid.ColumnSpan="2" />
```

4. Run the application and navigate to the question page. You should see question text for the first question.



Xamarin.Forms is a great choice when you're required to follow pixel-perfect mockups or leverage a lot of platform specific features.

Answer Text Default

5. Optional: If you try to launch the Question Page before the questions have finished loading you'll get a null reference exception. Add a public property to **HomePage.cs** to control the **IsEnabled** property of the start button. Set the property from the app's **OnStart** method: false when you begin the method and true once you have finished loading the questions. This requires several changes to your code, only complete this step if you have extra time.

14 Create and connect view models to drive the UI

In this part, you will create a view model to make it easier to control visual properties of the question page.

1. In the eXam PCL project, create a new class named **QuestionPageViewModel** and open the file.
    a. Make the class public
    b. Add or update the constructor signature to accept a **Game** object. Save the Game object in a field named **game**.
    c. In the constructor, call **Game**'s **Restart** method. This ensures that navigating to the question page will always start a new game.

```
public class QuestionPageViewModel
{
    private readonly Game game;

    public QuestionPageViewModel(Game game)
    {
        this.game = game;
        this.game.Restart();
    }
}
```

2. Open the **QuestionPageViewModel** file. Recall that the question page needs to display the text of the question and a correct/incorrect indicator once the user answers the question.
    a. Add a get-only string property named **Question**. Return **game.CurrentQuestion.Question** from the getter.

    b. Add a get-only string property named **Response**. The **Response** property requires a bit more code. First, do some error checking: test if **game.CurrentResponse** is null and, if so, return an empty string. Otherwise, you know the user has responded to the question and you can move on to checking if they responded correctly: compare **game.CurrentQuestion.Answer** and **game.CurrentResponse** to see if they responded correctly. Based on the comparison, return a string of your choice to indicate to the user whether they were correct or incorrect. *(continued on next page)*

```
public string Question
{
    get { return game.CurrentQuestion.Question; }
}

public string Response
{
    get
    {
        if (game.CurrentResponse == null)
        {
            return "";
        }

        return game.CurrentQuestion.Answer == game.CurrentResponse
            ? "Correct!"
            : "Incorrect";
    }
}
```

3. Open **QuestionPage.xaml.cs**
   a. Change the constructor signature that currently accepts a **QuizQuestion** to accept a **QuestionPageViewModel** instead. Assign the view model to the **BindingContext** of the page.

```
public QuestionPage(QuestionPageViewModel viewModel)
{
    this.BindingContext = viewModel;
    InitializeComponent();
}
```

4. Open **QuestionPage.xaml**.
   a. Find the label that gives the user the “Correct” / “Incorrect” feedback on their response. Bind the label’s **Text** property to the **Response** property of the view model.

```
<Label Text="{Binding Response}" Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" />
```

5. Open **HomePage.cs**.
   a. Locate the click-event handler for the start button.
   b. Change the instantiation of **QuestionPage**: create an instance of **QuestionPageViewModel**, pass it the **App.CurrentGame**, and pass the view model to the **QuestionPage** constructor.
   (continued on next page)

```
button.Clicked += async (sender, args) =>
{
    var viewModel = new QuestionPageViewModel(App.CurrentGame);
    await this.Navigation.PushAsync(new QuestionPage(viewModel));
};
```

6. Run the application and verify the question text is still displayed on the question page. The “Correct” / “Incorrect” response feedback will not be displayed yet.

15 Implement INotifyPropertyChanged

In this part, you will prepare your **QuestionPageViewModel** to update the UI when the user taps the True/False/Next buttons on the **QuestionPage**.

1. Open **QuestionPageViewModel.cs** in the PCL project.
   a. Implement **INotifyPropertyChanged** on the **QuestionPageViewModel** class: add the interface to the class declaration and add the public **PropertyChanged** event.

```
public class QuestionPageViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    // ...
}
```

   b. You need to update several visible properties on multiple views. In a case like this, it is reasonable to signal that all properties were updated instead of signalling the affected properties individually. Create a new **void** method named **RaiseAllPropertiesChanged**.
   (continued on next page)

c. In **RaiseAllPropertiesChanged**, you will raise the **PropertyChanged event**. First, ensure **PropertyChanged** is not null, and then raise the event. You'll need to instantiate a **PropertyChangedEventArgs** object, passing an empty string into its constructor (it is the empty string that indicates to the bindings that "all properties have changed" and will cause all bindings against the view model to refresh their UI). You will call **RaiseAllPropertiesChanged** in the next exercise.

```
private void RaiseAllPropertiesChanged()
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(""));
}
```

16 Use Commands to respond to button taps

In this part, you will use Commands to respond to button taps from the Question Page. Commands help you decouple the UI from the code-behind and are a core technique in data binding and the MVVM pattern. The page has three buttons (True, False, and Next) and you will use Command objects to respond to taps on each of them.

1. Open **QuestionPageViewModel.cs** in the PCL project.
   a. Add three public properties of type **Command** (from the Xamarin.Forms namespace), one for each button. Name them **TrueSelected**, **FalseSelected**, and **NextSelected**. You can use the C# "automatic property" syntax; you do not need to declare a backing field or implement get/set manually.

```
public class QuestionPageViewModel : INotifyPropertyChanged
{

    public Command TrueSelected { get; set; }
    public Command FalseSelected { get; set; }
    public Command NextSelected { get; set; }

    // ...
}
```

b. Create methods that implement the logic required when the user taps the True or False button: **void OnTrue()** and **void OnFalse()**:
   i. In the **OnTrue** method, call **game.OnTrue** and then invoke **RaiseAllPropertiesChanged**.
   ii. In the **OnFalse** method, call **game.OnFalse** and then invoke **RaiseAllPropertiesChanged**.
   iii. When the user selects their response, the Next button needs to become enabled. In both **OnTrue** and **OnFalse** call **NextSelected.ChangeCanExecute**. This tells the **NextSelected** Command to raise an event that notifies the Next button that its enabled state must be updated.

```
private void OnTrue()
{
    game.OnTrue();
    RaiseAllPropertiesChanged();
    NextSelected.ChangeCanExecute();
}

private void OnFalse()
{
    game.OnFalse();
    RaiseAllPropertiesChanged();
    NextSelected.ChangeCanExecute();
}
```

c. Locate the **QuestionPageViewModel** constructor.
   i. In the constructor, instantiate a **Command** object passing in **OnTrue** to its constructor. Assign the **Command** object to your **TrueSelected** property.
   ii. Add the analogous code for **FalseSelected**: instantiate a **Command** object passing in **OnFalse** and assign it to **FalseSelected** property.

```
public QuestionPageViewModel(Game game)
{
    // ...
    TrueSelected = new Command(OnTrue);
    FalseSelected = new Command(OnFalse);
}
```

d. Create an **void** method for the Next button named **OnNext**. Call **game.NextQuestion** and check the Boolean result. If it's **true** call **NextSelected.ChangeCanExecute** and **RaiseAllPropertiesChanged**.

```
private void OnNext()
{
    bool result = game.NextQuestion();
    if (result)
    {
        NextSelected.ChangeCanExecute();
        RaiseAllPropertiesChanged();
    }
}
```

e. The Next button should only be enabled after the user has responded to the question. Add a method named **OnCanExecuteNext** that returns a **bool** which indicates this: return false if the current response in the game class is null, otherwise return true.

```
private bool OnCanExecuteNext()
{
    return game.CurrentResponse.HasValue;
}
```

f. In the **QuestionPageViewModel** constructor, instantiate a **Command** object passing in **OnNext** and **OnCanExecuteNext**. Assign the **Command** object to your **NextSelected** property.

```
public QuestionPageViewModel(Game game)
{
    // ...
    NextSelected = new Command(OnNext, OnCanExecuteNext);
}
```

2. Open **QuestionPage.xaml**.
   a. For each button, bind the button's **Command** property to the appropriate command in the view model.

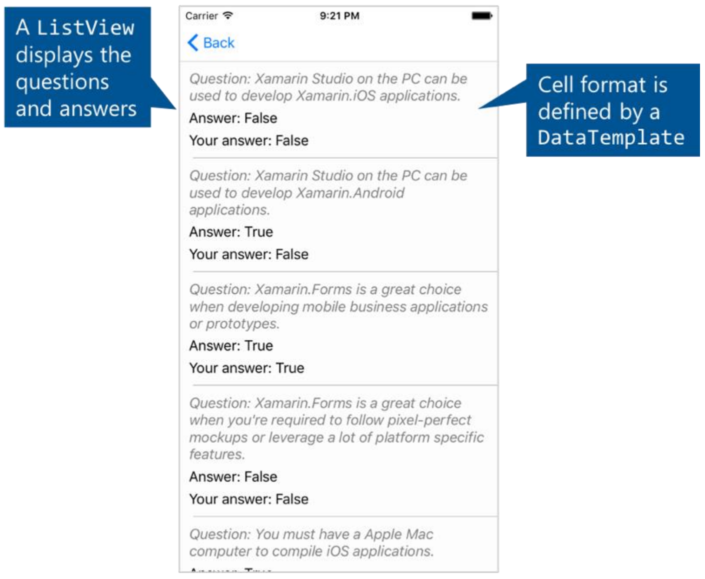
```
<Button Text="True" Grid.Row="1" Grid.Column="0" Command="{Binding TrueSelected}" />
<Button Text="False" Grid.Row="1" Grid.Column="1" Command="{Binding FalseSelected}" />
<Button Text="Next" Grid.Row="3" Grid.Column="1" Command="{Binding NextSelected}" />
```

3. Run the application and answer several questions. Notice the Next button is disabled until an answer is chosen.

4. Optional: Prevent the user from changing their response (for True/False question, it doesn’t make sense to let the user answer a question more than once). To do this, add a Boolean property to the view model to control the True and False buttons’ enabled properties. When the game’s current response is null, enable the True and False buttons. When the current response is not null, disable the True and False buttons.

17 Add a Page with a ListView

In this part, you will code a new Content Page with a ListView to display the results of the quiz. You will also add a Navigation class and two view models to complete your MVVM implementation. The UI for the new page should look similar to the screenshot below when you are done.


A ListView displays the questions and answers
Cell format is defined by a DataTemplate
Carrier
9:21 PM
Back

Question: Xamarin Studio on the PC can be used to develop Xamarin.iOS applications.
Answer: False
Your answer: False

Question: Xamarin Studio on the PC can be used to develop Xamarin.Android applications.
Answer: True
Your answer: False

Question: Xamarin.Forms is a great choice when developing mobile business applications or prototypes.
Answer: True
Your answer: True

Question: Xamarin.Forms is a great choice when you're required to follow pixel-perfect mockups or leverage a lot of platform specific features.
Answer: False
Your answer: False

Question: You must have a Apple Mac computer to compile iOS applications.

1. You will need a view model for the Quiz Question class. You have two options: add the provided code to the project or write it yourself.

   To add the provided code:
   a. Locate **QuizQuestionViewModel.cs** in the Assets folder and add it to the PCL. Take a moment to review the code. It is a simple class but it

demonstrates a few interesting patterns common to view models; for example, it provides a convenient **IsCorrect** property that is needed by the UI but isn't available directly in the underlying quiz-question data.

To write it yourself:

a. Add a new public class named **QuizQuestionViewModel** in the PCL project.
b. The view model will encapsulate a **QuizQuestion** instance as well as the user's response. Change the constructor signature to accept this data: a **QuizQuestion** and a Nullable bool (i.e. **bool?**).
c. Create a private field to store the passed-in **QuizQuestion** and a public property to hold the response. Assign both in the constructor.
d. Create read-only properties to expose the three public properties of **QuizQuestion** (Question, Answer, Explanation). Your properties in **QuizQuestionViewModel** can use the same names as the underlying **QuizQuestion** properties (Question, Answer, Explanation).
e. Create a read-only Boolean property named **IsCorrect** that determines whether the user's response is the correct answer for that question.

```
public class QuizQuestionViewModel
{
    public string Question { get { return quizQuestion.Question; } }
    public bool Answer { get { return quizQuestion.Answer; } }
    public string Explanation { get { return quizQuestion.Explanation; } }

    public bool? Response { get; private set; }

    public bool IsCorrect { get { return quizQuestion.Answer == Response; } }

    QuizQuestion quizQuestion;

    public QuizQuestionViewModel(QuizQuestion quizQuestion, bool? response)
    {
        this.Response = response;
        this.quizQuestion = quizQuestion;
    }
}
```

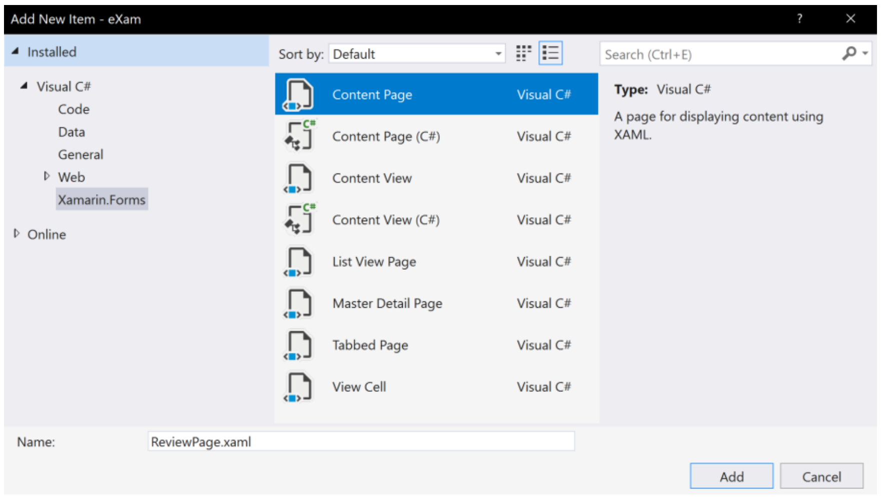
2. Add a new public class named **ReviewPageViewModel** to the PCL project and open the file.
   a. Add a public **List<QuizQuestionViewModel>** property named **QuestionViewModels**. This property will hold the data that the list view will display.
   b. Add/Change the constructor signature to accept a **Game** object named **game**.

c. Instantiate a **List<QuizQuestionViewModel>** and assign it to your property.

d. Create a **for** loop that runs from 0 to **game.NumberOfQuestions**. In each iteration, create a new **QuizQuestionViewModel** for one of the game's questions and add it to the **QuestionViewModels** property. Use the **Questions** and **Responses** properties of **game** to get the question and response needed to create the view model object.

```
public class ReviewPageViewModel
{
    private readonly Game game;
    public List<QuizQuestionViewModel> QuestionViewModels { get; set; }

    public ReviewPageViewModel(Game game)
    {
        this.game = game;
        QuestionViewModels = new List<QuizQuestionViewModel>();
        for (int index = 0; index < game.NumberOfQuestions; index++)
        {
            var question = game.Questions[index];
            var response = game.Responses[index];
            var viewModel = new QuizQuestionViewModel(question, response);
            QuestionViewModels.Add(viewModel);
        }
    }
}
```

3. Add a new XAML Content Page to the PCL project. Name it **ReviewPage**.



Name: ReviewPage.xaml

4. Open **ReviewPage.xaml.cs**.
    a. Change the constructor signature to accept a **ReviewPageViewModel**.
    b. Set the **BindingContext** of the page to the **ReviewPageViewModel**.

```
public partial class ReviewPage : ContentPage
{
    public ReviewPage(ReviewPageViewModel viewModel)
    {
        InitializeComponent();
        this.BindingContext = viewModel;
    }
}
```

5. Open **ReviewPage.xaml**
    a. Remove any default UI from the XAML.
    b. Add a **ListView** to the XAML page's Content.
    c. Use the **x:Name** directive to set the ListView's name to **listQuestions** and set **HasUnevenRows** to **true**.
    d. Bind the **ItemsSource** to **QuestionViewModels;** we added this property the view model earlier.
    e. Use the property-element syntax to assign the **ItemTemplate** property. Inside, create a **DataTemplate**.
    f. Finally, nest a custom cell inside the **DataTemplate** by creating a **ViewCell**.
    g. Inside the view cell, use a **StackLayout** to display three **Label**s. Bind each **Label**'s **Text** property to a property on a **QuizQuestionViewModel**: Question, Answer, and Response.
    h. Optionally, use **StringFormat** within the **Text** bindings to clarify the displayed text. For example **Text="{Binding Response, StringFormat='Your answer: {0}'}"**.

```
<ListView
    x:Name="listQuestions"
    HasUnevenRows="True"
    ItemsSource="{Binding QuestionViewModels}">

    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <StackLayout>
                    <Label Text="{Binding Question}" />
                    <Label Text="{Binding Answer}" />
                    <Label Text="{Binding Response, StringFormat='Your answer: {0}'}" />
                </StackLayout>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

6. Locate **NavigationService.cs** in the provided Assets folder and add it to the PCL. Take a moment to review the file. This will make it easy to navigate the app from code inside a view model without coupling the view model to the UI framework.

7. Open the Application class: **App.xaml.cs** .
    a. Register the navigation service in the constructor using the **DependencyService**'s static **Register** method. The method doesn't take any regular arguments, but it does require you to pass the type you are registering as the generics parameter.

```
public App()
{
    // ...
    DependencyService.Register<NavigationService>();
}
```

8. Open **QuestionPageViewModel.cs** and locate the **OnNext** method
    a. Retrieve an instance of the navigation service using the **DependencyService's** Get method
    b. If the **game.NextQuestion** returns false, use the navigation service's **GoToPageAsync** method to navigate to the review page by passing **AppPage.ReviewPage**. As this is a Task-based **async** method, you should **await** the call and add the **async** keyword to the **OnNext** method.

```
private async void OnNext()
{
    bool result = game.NextQuestion();
    if (result)
    {
        NextSelected.ChangeCanExecute();
        RaiseAllPropertiesChanged();
    }
    else
    {
        NavigationService navigationService =
DependencyService.Get<NavigationService>();
        await navigationService.GotoPageAsync(AppPage.ReviewPage);
    }
}
```

9. In **HomePage.cs**, update the start button's **Clicked** event handler to use the navigation service. The **NavigationService** class will handle creating the **QuestionPageViewModel** and passing it to the **QuestionPage**.

```
button.Clicked += async (sender, args) =>
{
    NavigationService navigationService =
DependencyService.Get<NavigationService>();
    await navigationService.GotoPageAsync(AppPage.QuestionPage);
};
```

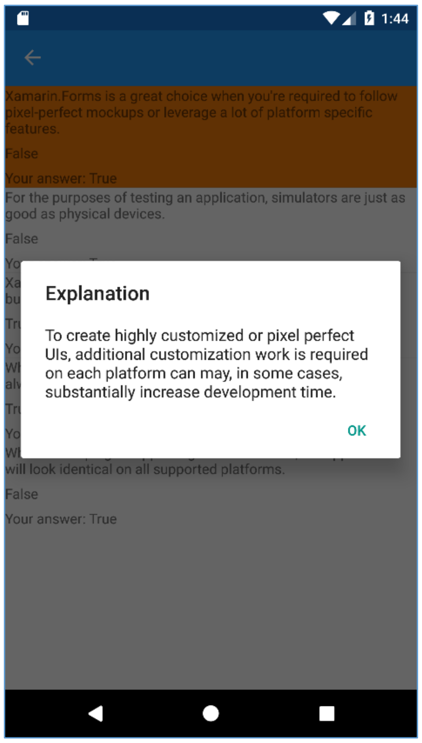
18 Display question explanation

In this part, you will respond to user taps on the Review Page's ListView to display an explanation of each question.

1. Open **ReviewPage.xaml.cs** in the PCL project.
    a. Subscribe to the list view's **ItemTapped** event using any subscription technique you prefer (named method, lambda, delegate).
    b. The **ItemTappedEventArgs** has a property named **Item** that is a reference to the object that is bound to the tapped cell. In our case, it's a **QuizQuestionViewModel**. Cast **e.Item** to a **QuizQuestionViewModel** and save it to a local variable named **qqvm**.
    c. Display an Alert Dialog using the Page's inherited **DisplayAlert** method. When displaying the alert, show the **Explanation** property of **qqvm**.

```
public ReviewPage(ReviewPageViewModel viewModel)
{
    InitializeComponent();
    this.BindingContext = viewModel;
    listQuestions.ItemTapped += async (sender, args) =>
    {
        QuizQuestionViewModel qqvm = args.Item as QuizQuestionViewModel;
        if (qqvm != null)
        {
            await this.DisplayAlert("Explanation", qqvm.Explanation, "Ok");
        }
    };
}
```

d. Run the application and tap on a cell in the Result Page's list view.



1:44

Xamarin.Forms is a great choice when you're required to follow pixel-perfect mockups or leverage a lot of platform specific features.

False

Your answer: True

For the purposes of testing an application, simulators are just as good as physical devices.

False

Yo

Xa
bu

Tr

Yo

Wh
alv

Tr

Yo

Wh
will look identical on all supported platforms.

False

Your answer: True

Explanation

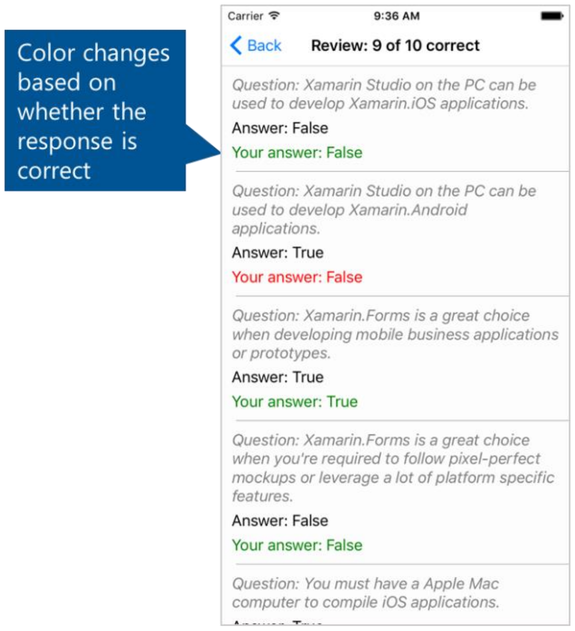
To create highly customized or pixel perfect UIs, additional customization work is required on each platform can may, in some cases, substantially increase development time.

OK

Congratulations! You have built a full Xamarin.Forms application that pulls data from an Azure Service, stores it locally in the app, and guides the user through a set of screens to answer questions.

19 Polish the results page using a DataTrigger

In this part, you will use a DataTrigger to change the text color of the user's response depending on whether the response is correct (see below).



Carrier 9:36 AM

Back Review: 9 of 10 correct

Question: Xamarin Studio on the PC can be used to develop Xamarin.iOS applications.
Answer: False
Your answer: False

Question: Xamarin Studio on the PC can be used to develop Xamarin.Android applications.
Answer: True
Your answer: False

Question: Xamarin.Forms is a great choice when developing mobile business applications or prototypes.
Answer: True
Your answer: True

Question: Xamarin.Forms is a great choice when you're required to follow pixel-perfect mockups or leverage a lot of platform specific features.
Answer: False
Your answer: False

Question: You must have a Apple Mac computer to compile iOS applications.

1. Open **ReviewPage.xaml** in the PCL project.
   a. Find the **Label** used to display the user's response and change the syntax to use a full **</Label>** closing tag.
   b. Set the **Label**'s **TextColor** to **Red**.
   c. Use property-element syntax to assign the **Triggers** property of the **Label**.
   d. Within the **Triggers** tag, add a **DataTrigger**. Use data binding to bind the **DataTrigger**'s **Binding** property to the **IsCorrect** property of the view model. Set the **Value** property to **True**. Set the **TargetType** to **Label.**
   e. Make sure the **DataTrigger** has a full closing tag.
   f. Inside the **DataTrigger** tags, add a **Setter** to update the label's **TextColor** (Property) to **Green** (Value).

```
<Label Text="{Binding Response, StringFormat='Your answer: {0}'}"
TextColor="Red">
    <Label.Triggers>
        <DataTrigger Binding="{Binding IsCorrect}" TargetType="Label"
Value="True" >
            <Setter Property="TextColor" Value="Green" />
        </DataTrigger>
    </Label.Triggers>
```

```
</Label>
```

2. Run the application and complete the quiz to display the Review Page.

20 Turn on cell recycling

In this part, you will enable cell recycling for the table view to reduce memory usage and improve performance.

0. Open **ReviewPage.xaml** in the PCL project.
   a. Set the list view's **CachingStrategy** to **RecycleElement.**

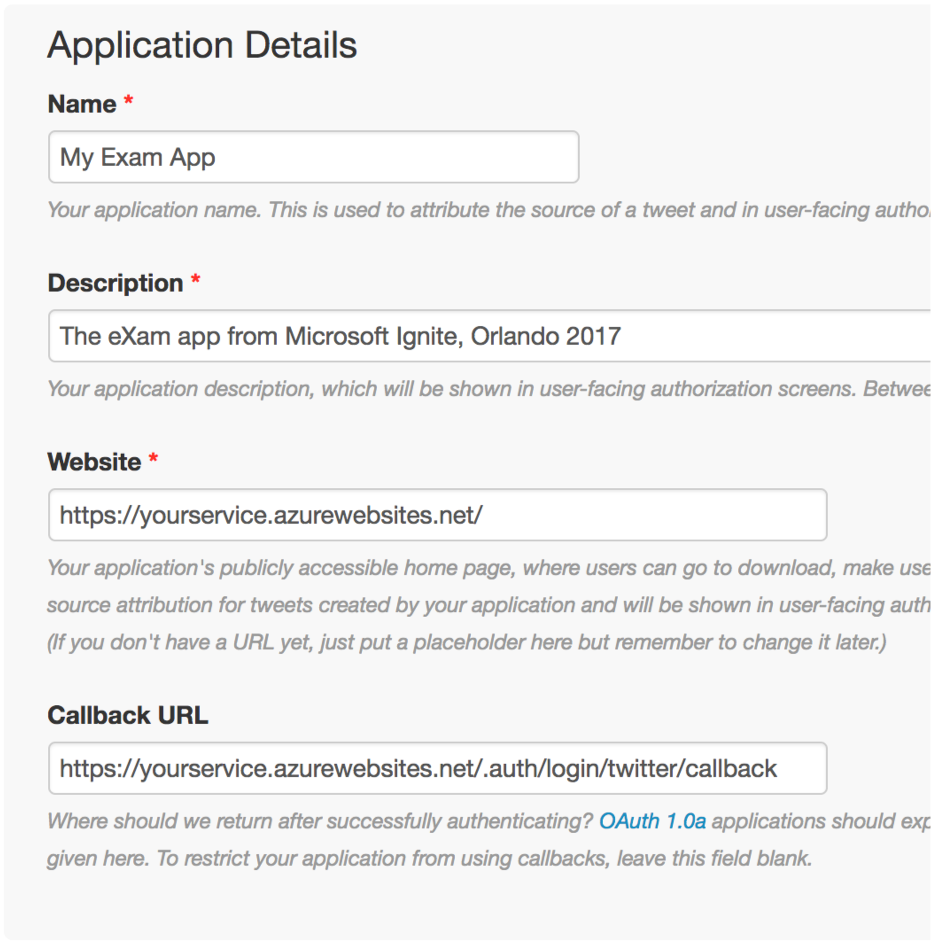
```
<ListView
    x:Name="listQuestions"
    HasUnevenRows="True"
    CachingStrategy="RecycleElement"
    ItemsSource="{Binding QuestionViewModels}">
```

1. Run the application. With this amount of data, you're unlikely to notice a performance difference, but the list view is now *reusing* the data template definitions and visual cells as you scroll rather than recreating them each time. For larger lists of data this can significantly improve performance.

21 Setup Azure Authentication

Go to your Azure portal page.

1. Turn on Authentication/Authorization to the ON Switch
2. Go to apps.twitter.com and navigate to MyApps
    a. Create an App – give it a name and description. If you do not have a Twitter account you will need to create one as part of the process
    b. For the website use your Azure Mobile App site
    c. For the Callback URL you will have to go back to Azure and configure your Twitter app with the callback https://<YOUR-APP-SERVICE-URL>.azurewebsites.net/.auth/login/twitter/callback
    d. Save your changes to your Twitter App. This will give you the API key and Secret. Click Manage keys to see them



Application Details

Name *

My Exam App

Your application name. This is used to attribute the source of a tweet and in user-facing autho

Description *

The eXam app from Microsoft Ignite, Orlando 2017

Your application description, which will be shown in user-facing authorization screens. Betwee

Website *

https://yourservice.azurewebsites.net/

Your application's publicly accessible home page, where users can go to download, make use source attribution for tweets created by your application and will be shown in user-facing auth (If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

https://yourservice.azurewebsites.net/.auth/login/twitter/callback

Where should we return after successfully authenticating? OAuth 1.0a applications should exp given here. To restrict your application from using callbacks, leave this field blank.

3. Go back to your Azure portal and Paste in the API Key and Secret in the Authorization configuration screen for your twitter app and **remember to Save**!

4. Test by going to the browser and entering the URL https://<YOUR-APP-SERVICE-URL>.azurewebsites.net/.auth/login/twitter.
    a. This should redirect you to the login page for your Twitter app and step you through the process. If you can't see the Twitter information, it is quite likely that you need to ensure your Twitter configuration matches.

Setup Authorization within the Xamarin.Forms app

5. First, we are going to need our **AzureURL** in a few places, so we will move it to an **AppConstants.cs** File

```
public class AppConstants
{
    public const string AzureUrl = "https://<YOUR-APP-SERVICE-
URL>.azurewebsites.net/";
}
```

6. Add a new public static class named **GameManager** in the **Services** folder in the PCL project
    a. Add a static async method named **CreateGameAsync** that returns **Task<Game>**
    b. Add another static async method named **LoadQuestionsAsync** that returns **Task<IEnumerable<ExamQuestion>>**.
    c. Move the code from the **App.xaml.cs OnCreate()** to this method and return await **Task.Run(() => JsonConvert.DeserializeObject<List<ExamQuestion>>(questionsText));**
    d. Back in **CreateGameAsync**
        i. Call this method to get questions
        ii. Return a new game passing in to the constructor **questions.ToList()**.

Your **GameManager.cs** file should then look like the following:

```
using Newtonsoft.Json;
using Plugin.Connectivity;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;
using System.Threading.Tasks;
using Xamarin.Forms;

namespace eXam.Services
{
    public static class GameManager
    {
        public static async Task<Game> CreateGameAsync()
        {
            var questions = await LoadQuestionsAsync();
            return new Game(questions.ToList());
        }

        static async Task<IEnumerable<QuizQuestion>> LoadQuestionsAsync()
        {
            IFileHelper fileHelper = DependencyService.Get<IFileHelper>();
            string cachedQuestions;

            if (CrossConnectivity.Current.IsConnected)
            {
                // We're online
                IQuestionService questionService = new AzureQuestionService();
                var questionsList = await questionService.GetQuestionsAsync();
                cachedQuestions = JsonConvert.SerializeObject(questionsList);
            }
            else
            {
                // We're offline.  Load the questions from the local cache
                cachedQuestions = await
fileHelper.LoadLocalFileAsync("cachedquestions.json");
                if (string.IsNullOrWhiteSpace(cachedQuestions))
                {
                    var assembly = typeof(App).GetTypeInfo().Assembly;
                    using (var resourceStream =
assembly.GetManifestResourceStream("eXam.Data.questions.json"))
                    {
                        using (var reader = new StreamReader(resourceStream))
                        {
                            cachedQuestions = reader.ReadToEnd();
                        }
                    }
                }
            }

            var embeddedQuestions =
JsonConvert.DeserializeObject<List<QuizQuestion>>(cachedQuestions);

            await fileHelper.SaveLocalFileAsync("cachedquestions.json",
cachedQuestions);

            return embeddedQuestions;
        }
    }
}
```

We need to add local storage for our authentication. There is a Nuget package that lets use this in a cross-platform way.

7. Add the NuGet package **sameerIOTApps.Plugin.SecureStorage** to all of your client-side projects
8. Add a class named **AuthStorage** to the PCL and add the following using clauses:

```
using Microsoft.WindowsAzure.MobileServices;
using Plugin.SecureStorage;
using System;

public class AuthStorage
{
}
```

9. Add two constant strings to the **AuthStorage** class. Add one for **userIdKey** and another for **tokenKey**. Set them to ":UserId" and ":Token" respectively.

```
const string userIdKey = ":UserId";
const string tokenKey = ":Token";
```

We need three other members of the class: a static boolean property **HasLoggedIn**, a static void method **LoadSavedUserDetails()** that takes a **MobileServiceClient** as a parameter, and a static void method **SaveUserDetails()** that takes a **MobileServiceClient** as a parameter.

10. Add a **HasLoggedIn** with a getter only. Make sure this property returns true if we have a userIdKey and a TokenKey. We can get this by checking **CrossSecureStorage.Current.HasKey**.

```
public static bool HasLoggedIn
{
    get
    {
        return (CrossSecureStorage.Current.HasKey(userIdKey)
                && CrossSecureStorage.Current.HasKey(tokenKey));
    }
}
```

11. Add the **SaveUserDetails()** method and use **CrossSecureStorage.Current.SetValue()** to set the UserIdKey and TokenKey. You can obtain these from **client.CurrentUser**

```
public static void SaveUserDetails(MobileServiceClient client)
{
    CrossSecureStorage.Current.SetValue(userIdKey, client.CurrentUser.UserId);
    CrossSecureStorage.Current.SetValue(tokenKey,
        client.CurrentUser.MobileServiceAuthenticationToken);
}
```

12. Add the **LoadSavedUserDetails()** method. It should check **HasLoggedIn**, and if we are not logged in, throw a **new ArgumentException("You do not have saved credentials")**. Get the values for userId and token from **CrossSecureStorage.Current.GetValue()**. Once they have been loaded, update the client to set its **CurrentUser** to a new **MobileServiceUser** that uses these values.

```
public static void LoadSavedUserDetails(MobileServiceClient client)
{
    if (!HasLoggedIn)
        throw new ArgumentException("You do not have saved credentials");

    string userId = CrossSecureStorage.Current.GetValue(userIdKey);
    string token = CrossSecureStorage.Current.GetValue(tokenKey);

    client.CurrentUser = new MobileServiceUser(userId)
    {
        MobileServiceAuthenticationToken = token
    };
}
```

When it is all put together, you should have something that looks like the following:

```
using Microsoft.WindowsAzure.MobileServices;
using Plugin.SecureStorage;
using System;

namespace eXam
{
    public class AuthStorage
    {
        const string userIdKey = ":UserId";
        const string tokenKey = ":Token";

        public static bool HasLoggedIn
        {
            get
            {
                return (CrossSecureStorage.Current.HasKey(userIdKey)
                        && CrossSecureStorage.Current.HasKey(tokenKey));
            }
        }
        public static void LoadSavedUserDetails(MobileServiceClient client)
        {
            if (!HasLoggedIn)
                throw new ArgumentException("You do not have saved credentials");

            string userId = CrossSecureStorage.Current.GetValue(userIdKey);
            string token = CrossSecureStorage.Current.GetValue(tokenKey);

            client.CurrentUser = new MobileServiceUser(userId)
            {
                MobileServiceAuthenticationToken = token
            };
        }

        public static void SaveUserDetails(MobileServiceClient client)
        {
            CrossSecureStorage.Current.SetValue(userIdKey,
client.CurrentUser.UserId);
            CrossSecureStorage.Current.SetValue(tokenKey,
                client.CurrentUser.MobileServiceAuthenticationToken);
        }
    }
}
```

13. In your Android host project add to the **MainActivity.cs** file the namespace reference **using Plugin.SecureStorage;**
    a. In the same **MainActivity.cs** file in the **OnCreate()** method, set the **SecureStorageImplementation.StoragePassword = Build.Id;**

```
protected override void OnCreate(Bundle bundle)
{
    SecureStorageImplementation.StoragePassword = Build.Id;
    // ...
}
```

14. In your Windows UWP host project add to the **App.xaml.cs** file the namespace reference **using Plugin.SecureStorage;**
    a. In the same **App.xaml.cs** file, in the **OnLaunched()** method set the **WinSecureStorageBase.StoragePassword = Package.Current.Id.ToString();**

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    // ...
    WinSecureStorageBase.StoragePassword = Package.Current.Id.ToString();
    // ...
}
```

22 Add Platform Specific code for Azure login

In this solution, our shared code lives in a Portable Class Library. To use platform specific features within a portable class library, like our platform specific Login we will need an interface.

1. Add the existing file called **IAzurePlatformLogin.cs** to the Interfaces folder. It is in the Assets folder you were given.

2. In each of the platform specific projects create a **Services** Folder and add a class that implements that interface and register it with the **DependencyService**.

To the iOS host project, create and add the following file in the **Services** folder and name it **iOSAzurePlatformLogin.cs**:

```
using UIKit;
using Microsoft.WindowsAzure.MobileServices;
using Xamarin.Forms;
using eXam.iOS.Services;
using System.Threading.Tasks;

[assembly: Dependency(typeof(iOSAzurePlatformLogin))]

namespace eXam.iOS.Services
{
    public class iOSAzurePlatformLogin : IAzurePlatformLogin
    {
        public async Task<MobileServiceUser> PerformLogin(MobileServiceClient 
client, MobileServiceAuthenticationProvider provider)
        {
            return await 
client.LoginAsync(UIApplication.SharedApplication.KeyWindow.RootViewController, 
provider);
        }
    }
}
```

In the Android host project create and add the following file in the **Services** folder and name it **DroidAzurePlatformLogin.cs**:

```
using Microsoft.WindowsAzure.MobileServices;
using Xamarin.Forms;
using eXam.Droid.Services;
using System.Threading.Tasks;

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

```
namespace eXam.Droid.Services
{
    public class DroidAzurePlatformLogin : IAzurePlatformLogin
    {
        public async Task<MobileServiceUser> PerformLogin(MobileServiceClient
client, MobileServiceAuthenticationProvider provider)
        {
            return await client.LoginAsync(Forms.Context, provider);
        }
    }
}
```

In the UWP host project create and add the following file in the **Services** folder and name it **UWPAzurePlatformLogin.cs**:

```
using Microsoft.WindowsAzure.MobileServices;
using Xamarin.Forms;
using eXam.UWP.Services;
using System.Threading.Tasks;

[assembly: Dependency(typeof(UWPAzurePlatformLogin))]

namespace eXam.UWP.Services
{
    class UWPAzurePlatformLogin : IAzurePlatformLogin
    {
        public async Task<MobileServiceUser> PerformLogin(
           MobileServiceClient client, MobileServiceAuthenticationProvider
provider)
        {
            return await client.LoginAsync(provider);
        }
    }
}
```

2. Add the **SigninPage.xaml, SigninPage.xaml.cs**, **LoadingQuestionsPage.xaml, LoadingQuestionsPage.xaml.cs** from the Assets folder to the **eXam** PCL project. Set the Build Action of the XAML pages to EmbeddedResource. Take a look through the files.
    a. The **SigninPage.cs** uses the **AuthStorage** utility we created and the **IAzurePlatformLogin** to Sign in.
    b. The **LoadingQuestionsPage.c**s page is a basic screen to show the user that the questions are loading in case the load operation takes a long time.
3. Change the navigation to support the new pages
    a. Back in the **App.xaml.cs**, create a static instance of **MobileServiceClient** and a static **HomePage** variable
    b. Add two new async void methods: **NavigateToExamPage()** and **NavigateToSigninPage()**

c. In **NavigateToExamPage()**, first set the **Application**'s **MainPage** to a new **LoadingQuestionsPage**, then set the **CurrentGame** using the **GameManager**.
d. Set **HomePage** to a new **HomePage** and set its **IsStartButtonEnabled** to true.
e. Set the **MainPage** to a new **NavigationPage** passing in the **HomePage**.
f. In **NavigateToSignInPage()** set the **MainPage** to a new **SigninPage** passing in the client.
g. In the constructor, check to see if the user has logged in using our **AuthStorage** class. If they have we can load those details and start the exam page by using the **NavigateToExamPage()** method. If the user hasn't logged in before, we'll take them directly to the **SigninPage**.

Your **App.xaml.cs** file should then look like the following:

```
using eXam.Services;
using Microsoft.WindowsAzure.MobileServices;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
namespace eXam
{
    public partial class App : Application
    {
        static MobileServiceClient client = new
MobileServiceClient(AppConstants.AzureUrl);

        public App()
        {
            InitializeComponent();
            DependencyService.Register<NavigationService>();
            DependencyService.Register<AzureQuestionService>();

            if (AuthStorage.HasLoggedIn)
            {
                AuthStorage.LoadSavedUserDetails(client);

                App.NavigateToExamPage();
            }
            else
            {
                App.NavigateToSigninPage();
            }
        }

        public static async void NavigateToExamPage()
        {
            Current.MainPage = new LoadingQuestionsPage();
            CurrentGame = await GameManager.CreateGameAsync();

            Current.MainPage = new NavigationPage(new HomePage());
        }

        public static void NavigateToSigninPage()
        {
            Current.MainPage = new SigninPage(client);
        }

        public static Game CurrentGame { get; set; }
    }
}
```

You’re now ready to run your application and authenticate through the Azure Mobile Services.

Optional Ideas

Congratulations! You have built a full Xamarin.Forms application that pulls data from a web endpoint, stores it locally in the app, and guides the user through a set of screens to answer questions!

Here are some ideas to continue extending the application while you are here at Evolve. Feel free to expand on this list and ask questions as you continue your learning!

- Every Xamarin.Forms page has a built in **Title** property that you can set. When displayed within a **NavigationPage**, the title will be shown on the **NavigationBar**. Set the **Title** for the Review page so that it shows the number of correct answers.
- Set a custom background color on the Question page or the review page – what else needs to be considered? Try running it on multiple platforms.
- Disable the Start **Button** at application start-up until the game engine is instantiated.
- Add icons to each native head project to display a customized icon in the launch screen for each mobile platform.

- Xamarin.Forms' Navigation page has a built in Back button; its default text is "Back". Each page can specify what text is displayed when it is the target of the back operation. For the **HomePage**, set the text of the Back button from its default of "Back" to something more meaningful such as "End Game" using **NavigationPage.SetBackButton**Title.