workshop overview

# workshop goals

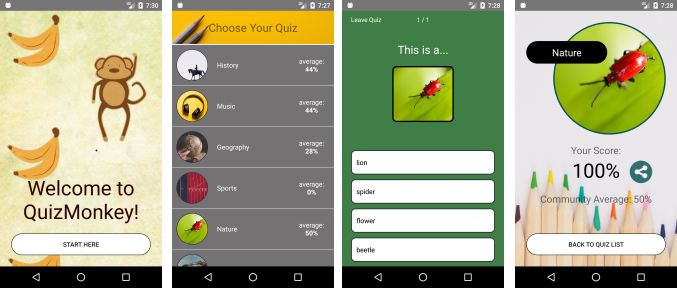
In this workshop you'll build a real NativeScript App based on Javascript. You'll see how such an app can be structured, how to handle a flow of information in the app and how to affect the views and navigation in it.

# app description and flow

The app you'll be building is called QuizMonkey. It'll hold a list of quizzes and when a user chooses a quiz they'll go through some questions and then see a summary of how well they did.

These are the views you'll be creating:

Welcome View Quiz List Question View Quiz Summary



Before starting

# basic installations

To Install NativeScript and all its prerequisites, follow the guide in this link:

<https://docs.nativescript.org/start/quick-setup>

# WIN/Linux users: create a functional Android Emulator

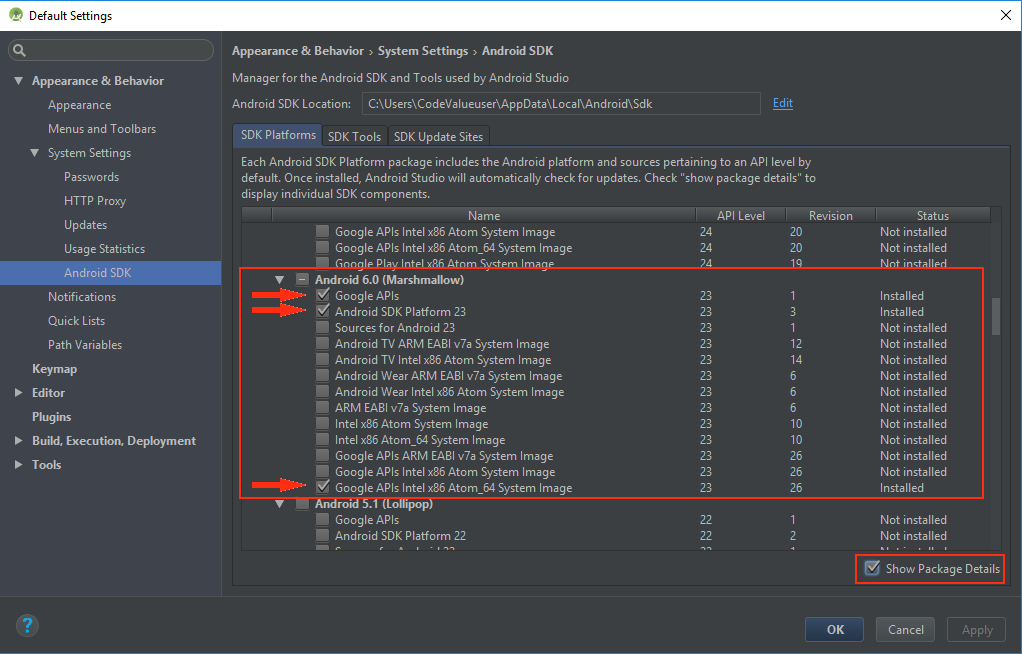
First, open AndroidStudio. This should have been installed for you with the basic NativeScript Installation.

##### Update Android SDK Packages

Go to Tools 🡪 Android 🡪 SDK Manager

Choose a platform package. Marshmallow is preferred as it seems to have less compatibility issues. Check the 'Show Package Details' checkbox and make sure you have the following components installed:

* Google APIs
* Android SDK Platform
* Google APIs Intel x86 Atom\_64 System Image

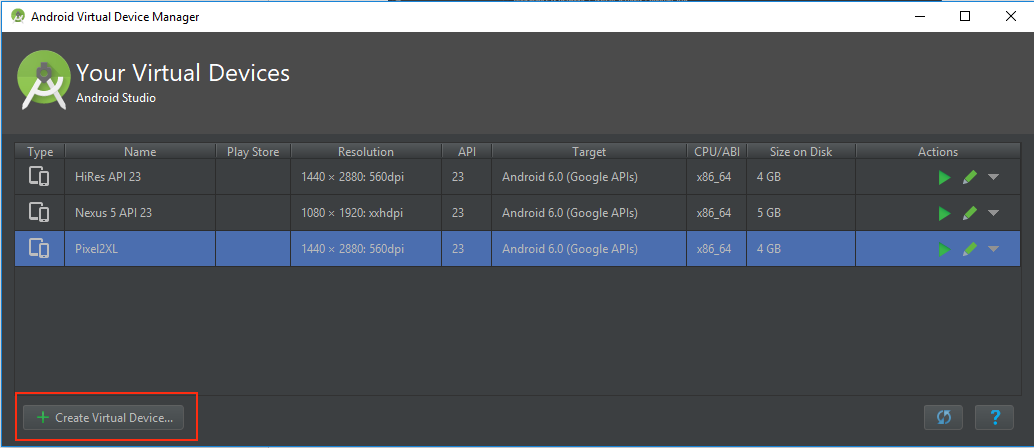


Click OK/Apply and let the wizard install the packages. This takes a few minutes.

##### Create a virtual device

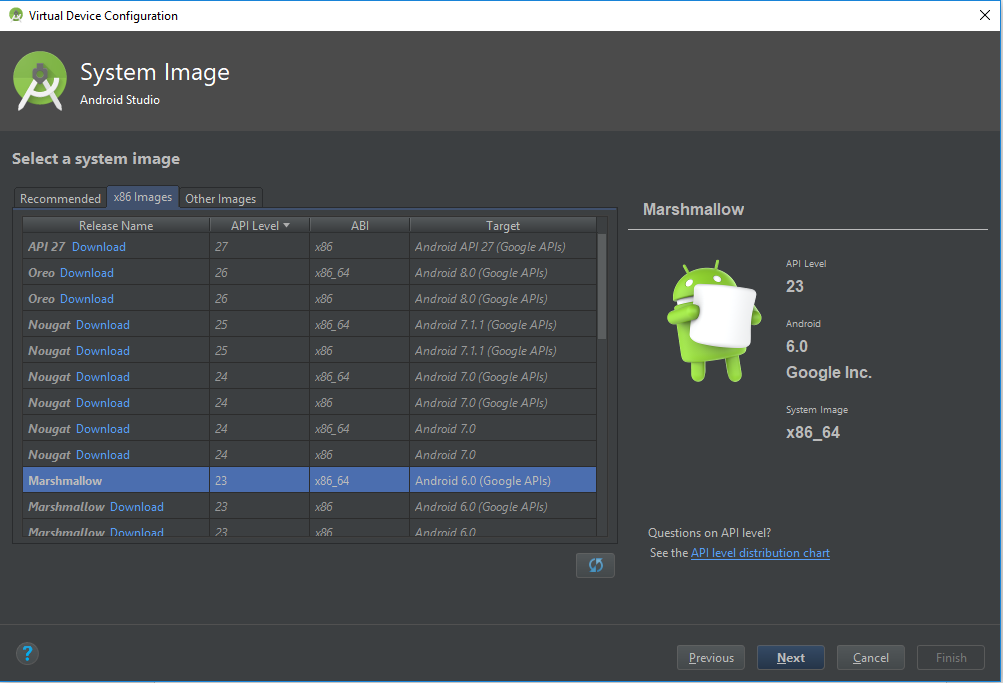
Go to Tools 🡪 Android 🡪 AVD Manager

Click Create Virtual Device



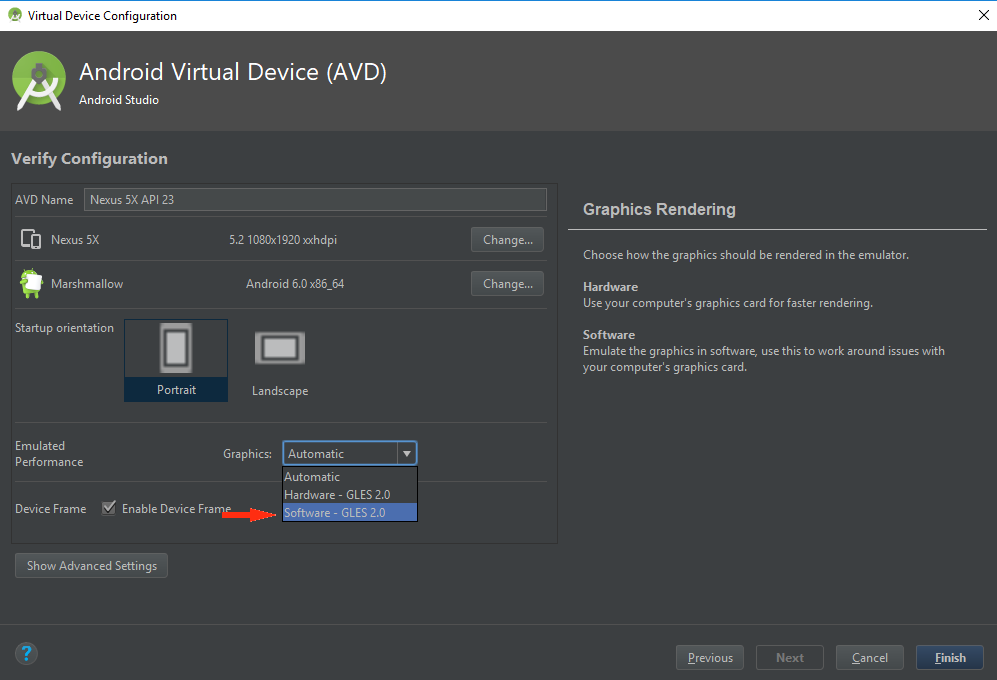
Select one of the phone models (a convenient size is a Nexus 5) and hit Next.

Select a system image that has Google APIs installed. Despite what the wizard sometimes recommends, we found it can be difficult to have devices from the first 'recommended' tab to have a status of authorized as devices for debugging, and they also prevent further configuration customizing. We recommend using this:



Click Next.

Name your device, and change the Emulated Performance Graphics settings from 'Automatic' to 'Software – GLES 2.0'. This is required to properly run GoogleMaps.



Click Finish.

To run the emulator press the play button next to it on the virtual devices list.

#### possible error: hyper-V on

To use the emulator, Hyper-V must be disabled. If you run into problems running it this might be the cause (especially since running Docker requires Hyper-V to be on). To turn Hyper-V off:

* Start a command prompt as Administrator.
* Run this command: bcdedit /set hypervisorlaunchtype off
* Reboot.

#### Tip: running the emulator without AndroidStudio

To run the emulator without starting AndroidStudio you can start a command line and cd to the folder containing the emulators. This would normally be under:

C:\Users\**YourUserFolder**\AppData\Local\Android\Sdk\Tools

Type:

emulator -avd Your\_Device\_Name.

If your device name has spaces in it, replace them with underscores.

getting started

# create your app

Open a command line window and CD to where you'd like to create your app.

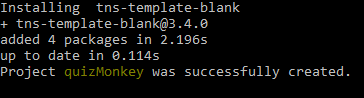
Start your app by creating a blank template: tns create yourAppName --template tns-template-blank

in our case – quizMonkey – it's going to be:

tns create quizMonkey --template tns-template-blank



If all goes well you should see a message similar to this:



A new folder by your app's name will be created at the containing folder, and it will hold the application skeleton.

# Test Your App and settings

Make sure your new application can run and that your environment is all set up:

Start your emulator/simulator.

CD to your newly created folder:



##### Android:

tns run android



This should also work: tns run android --emulator



##### iOS:

tns run ios



If the app runs successfully you will see something like this:



Start your IDE and see that you can affect it with live-reloading:

Look for the file 'home-page.xml' and change the label text there from "Home" to something else.

Save and look for the change in your emulator.

Note: sometimes hot-reloading does not work when changing xml/css files, in these cases you can make a small change on a js file (remove a semicolon, for example) to prompt it.

If you can see the app running, you're now all set to go!

# Organize your app

##### Add Folders

Add the following folders to your 'app' folder. You'll be using them later:

* views
* shared
* in the shared folder – create a 'view-models' folder
* mockData

You can delete the 'home' folder and its contents.

##### Add files

Add the files from the start-up package to the following folders:

* paste the two mock\*.json files to the app/mockData folder you've just created
* Paste the 'map-style.json' file to the ' app/tools/assets' folder
* Paste the contents of the folder named 'android images' to the folder 'app\App\_Resources\Android'.
* Paste the contents of the folder named 'ios images' to the folder 'app\App\_Resources\iOS'.

The image folders contain images in different sizes, to fit different devices and resolutions. They are then used as resources in the app. The app then automatically loads the best fit for it according to the device on which it is running, and so be more efficient on low-res devices without harming resolution on high-res devices. Images held this way are then referenced as resources in the application. You could also use normal image files and reference them accordingly when needed.

#### Tip: creating image resources online

There's an online tool to quite easily create the different images:

<http://nsimage.brosteins.com/>

# Optional: Disable landscape mode for the app

The app you'll be building will be best viewed in portrait mode. To disable landscape auto-rotating of the app, follow the steps for the different operating systems:

#### Android

Find the file named 'AndroidManifest.xml', located in 'app/App\_Resources/Android'. Paste the following line inside manifest 🡪 application 🡪 activity:

android:screenOrientation="portrait"

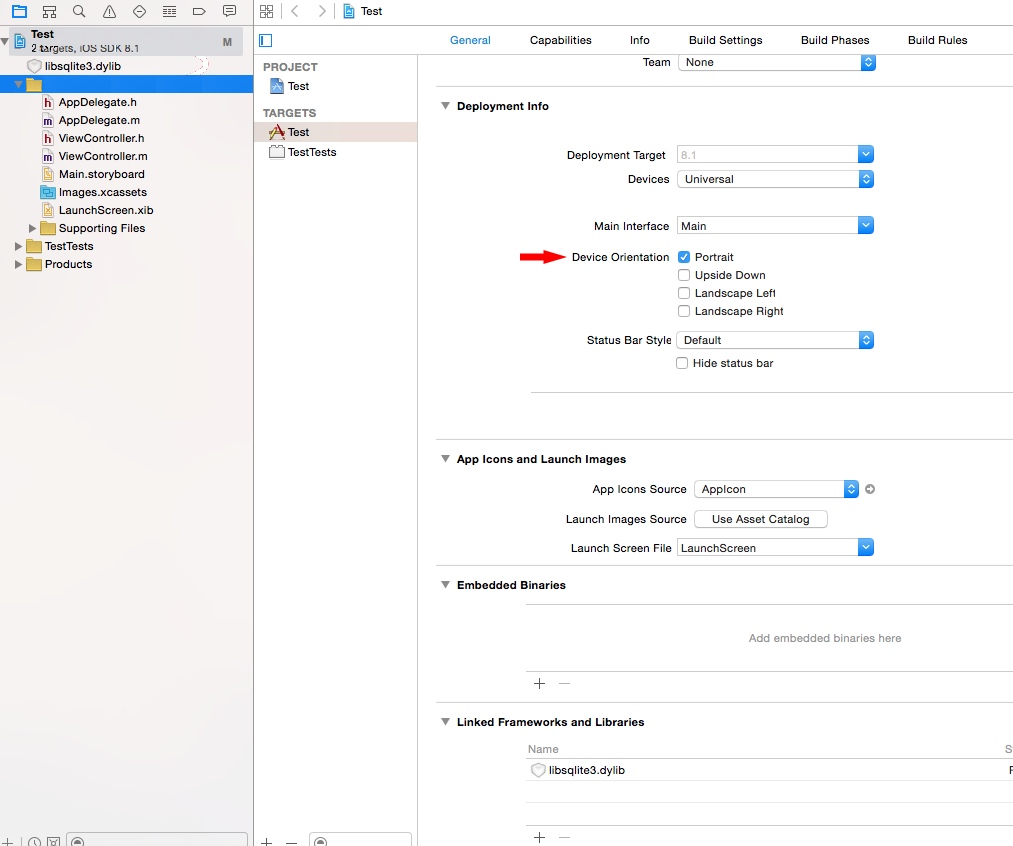
#### iOS

Start Xcode.

Select your project on the project navigator.

Select the target in the middle panel.

In General 🡪 Deployment Info 🡪 Device Orientation, tick only the 'Portrait' check box.



create the welcome view

# basic setup – xml file

Create a folder in the 'app/views' folder and name it 'welcome-view'.

In it create a file called 'welcome-view.xml'.

Paste this into this file:

<Page xmlns="http://schemas.nativescript.org/tns.xsd">

<Label text="Welcome to Quiz Monkey!"></Label>

</Page>

To make this view the start-up view for the app, open the file 'app.js' and change the path of the application.start module to your new file:

application.start({ moduleName: "/views/welcome-view/welcome-view" });

After saving, you'll see the label text you've written, and you'll also see an action bar holding your app name. To remove it, add this property to your page in the welcome-view.xml:

actionBarHidden="true"

This means making the .xml file look like this:

<Page xmlns="http://schemas.nativescript.org/tns.xsd"

actionBarHidden="true">

<Label text="Welcome to Quiz Monkey!"></Label>

</Page>

# Hook up to the view's loaded event from code-behind

Each UI element in the xml has a 'loaded' event you can listen to.

To set this up for the page loading event, add this property to the page in your xml:

loaded="onPageLoaded"

Create a new file in the same folder named 'welcome-view.js'. This will be the code-behind for the welcome-view.xml view. The code behind is where we'll place our view's logic and set up the data context for the page.

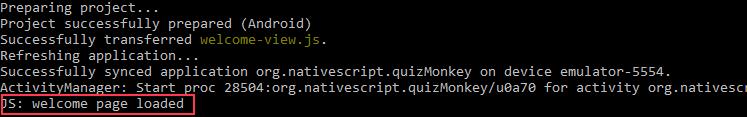
In the new file, add and export a function called 'onPageLoaded' and have it log something to the console:

*exports*.onPageLoaded = *function*() {

*console*.log("welcome page loaded");

};

After saving, you'll see your log in the terminal:



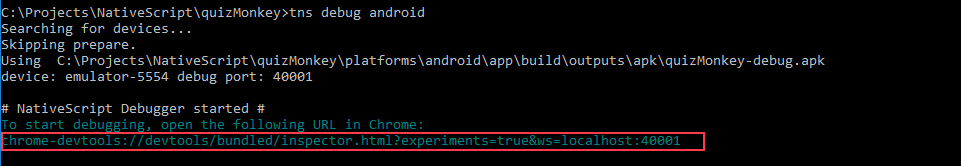
# Debugging

You can start your application in debug mode, and then you'll be able to see in the browser your calls to the console and debug your files using breakpoints, similar to normal web debugging:

Instead of using the run command in your cmd, use the debug command:

tns debug android

You will receive a link you can copy/paste into chrome:



Doing so will launch chrome devtools and connect it to your app.

You can also attach the debugger to a running application using

tns debug android --start

Visual Studio Code users can also debug using their IDE. See [here](https://docs.nativescript.org/tooling/debugging#debugging-with-visual-studio-code) for more details.

# Add elements to the View

Add a 'Start Here' button:

To have more than one element in a view you need to organize the elements in one of the [layout containers](https://docs.nativescript.org/ui/layout-containers) offered by NativeScript.

Here, you'll use the StackLayout. By default it stacks items vertically, but it can also stack them horizontally by specifying its orientation to vertical.

Add the StackLayout element to the root of the Page, and place in it the label and a new button:

<Button text="Start Here"/>

Your .xml file should now look similar to this:

<Page xmlns="http://schemas.nativescript.org/tns.xsd"

loaded="onPageLoaded"

actionBarHidden="true">

<StackLayout orientation="vertical">

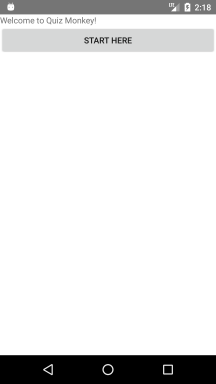
<Label text="Welcome to Quiz Monkey!"/>

<Button text="Start Here"/>

</StackLayout>

</Page>

And your page like this:



Add an event handler to the button tap event by declaring it on your button like this:

tap="onButtonTapped"

In the code-behind file, add and export an 'onTap' function, which will only write to the console for now:

*exports*.onButtonTapped = *function*() {

*console*.log("start button tapped");

};

# Add styles

This can be a good time to add some styling to the app.

Add class assignments to your xml elements:

<Page xmlns="http://schemas.nativescript.org/tns.xsd"

loaded="onPageLoaded"

actionBarHidden="true"

class="welcome-pageContainer imageBackgroundContainer">

<StackLayout class="welcome-stackLayoutContainer"

orientation="vertical">

<Label text="Welcome to Quiz Monkey!"

class="welcome-text"/>

<Button text="Start Here"

class="navigationButton"

onTap="onButtonTapped"/>

</StackLayout>

</Page>

Copy and paste the following to a new file you'll create: 'welcome-view.css', in the same folder:

.welcome-pageContainer{

*padding*: 25;

*background-image*: url("res://monkeybackground");

}

.welcome-text{

*font-size*: 45;

*vertical-align*: center;

*text-align*: center;

*white-space*: normal;

*margin-bottom*: 20;

*color*: rgb(49, 2, 2);

}

.welcome-stackLayoutContainer{

*vertical-align*: bottom;

}

As you can see, styling is very similar to normal web CSS styling, as NativeScript uses a subset of CSS. It is also using similar selectors.

The following are styles you'll be using again in the app, so paste them in the 'app.css' file which holds style common to the entire app:

.imageBackgroundContainer{

*background-repeat*: no-repeat;

*background-position*: center;

*background-size*: cover;

}

.navigationButton{

*background-color*: white;

*border-color*: black;

*border-width*: 1;

*border-radius*: 50%;

}

You should now have a nice-looking non-functional welcome view:



Create the quiz-list view

In this step, you'll put aside the welcome view for now, and create the quiz-list view. You will later be able to navigate to the quiz-list view through the welcome-view.

# Create and initialize the view's files

In the 'views' folder, create a new folder and name it 'quiz-list-view'. In it create a 'quiz-list-view.xml' file, a 'quiz-list-view.js' file and a 'quiz-list-view.css' file.

Specify this as the application's startup point, as you did before to the welcome-view. You will change this later.

Start by building your quiz-list-view.xml file:

<Page xmlns="http://schemas.nativescript.org/tns.xsd"

actionBarHidden="true"

loaded="onQuizListPageLoaded">

<GridLayout rows="auto, \*">

<Label text="Choose Your Quiz" />

<ListView items="{{ quizList }}" row="1">

</ListView>

</GridLayout>

</Page>

Note that:

* Here, the GridLayout container is used, and the "rows" property sets it to have two rows. The first one, set to 'auto', will set its height automatically to fit the height its children require. The second one, set to '\*', will take up the remaining available space.

Since no columns property is specified it will have by default one column.  
For the GridLayout children, column and row are set to 0 by default (in a zero-based count). The 'row="1"' assignment of the ListView puts it on the second row.

* The "{{XXX}}" syntax signals binding to the page's data context, which is not yet declared. Here, the ListView 'items' property is binded to a 'quizList' which you will soon define.

# Create and initialize a Quiz-List View Model

In the 'view-models' folder, add a 'quiz-list-view-model.js' file.

View-models in NativeScript are based on an 'observable' object, which helps update the app views as needed (hence its name). Here, since you need to bind your data to a collection of quizzes, you can utilize the ObservableArray module.

First, require the ObservableArray module at the top of the file, along with the mock quizzes data file:

*var* ObservableArray = require("data/observable-array").ObservableArray;

*var* mockQuizzesData = require("../../mockData/mockQuizzesData.json");

Then, instantiate the view model:

*function* QuizListViewModel(*items*) {

*var* viewModel = new ObservableArray(items);

//paste viewModel functions here

return viewModel;

}

*module*.*exports* = QuizListViewModel;

Paste the following three functions inside the view model:

addQuizzesToViewModel = *function* (*quizzes*) {

for (i = 0; i < quizzes.length; i++) {

viewModel.push({

id: quizzes[i].\_id,

name: quizzes[i].name,

image: quizzes[i].image,

});

};

}

loadMockDataQuizzes = *function* () {

addQuizzesToViewModel(mockQuizzesData);

}

viewModel.loadQuizzes = *function* () {

loadMockDataQuizzes();

}

The 'loadQuizzes' function will currently just call the 'loadMockDataQuizzes' function, which in turn takes the local mock data and passes it to a function that pushes its items to the view model's array.

# Connect the view-model to the code-behind

In the quiz-list-view.js file, require the view model you built and instantiate it:

*var* QuizListViewModel = require('../../shared/view-models/quiz-list-view-model');

*var* vm = new QuizListViewModel();

Although you can bind the page to the view-model directly, here, make the quizListViewModel part of another observable which will be the binding context for the view. Do this so that when you want to bind your page to other properties they do not have to be a part of the view-model:

First require the observableModule at the top of the file:

*var* observableModule = require("data/observable");

And then add this:

*var* quizListData = new observableModule.fromObject({

quizList: vm

})

This sets the view model you've created to be accessed under the name 'quizList'. This is what the ListView for the quizzes is binded to, in the .xml file.

Finally, upon page loading, set 'quizListData' to be the binding context for the page and load the quizzes in the view model by calling the 'loadQuizzes' function:

*exports*.onQuizListPageLoaded = *function* (*args*) {

page = args.object;

page.bindingContext = quizListData;

vm.loadQuizzes();

};

You should be getting something like this:



# Add an Item Template to the list of quizzes

The ListView holds six items, like the list it is binded to, but as you did not specify an item template for the items in the list they are not displayed coherently. Put this code inside the ListView:

<ListView.itemTemplate>

<GridLayout columns="auto, \*">

<Image src="{{image ? image : 'res://unknownquizimage'}}"

stretch="aspectFill"/>

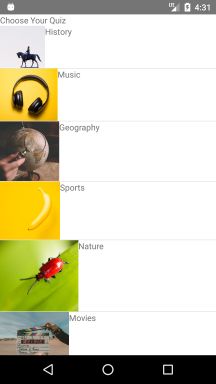
<Label text="{{name}}" col="1"/>

</GridLayout>

</ListView.itemTemplate>

The source for the image here is conditional, using a ternary experssion: it gives a fallback value for the image, just in case it was not specified in the data (as is the case for the 'sports' quiz).

This will translate to:



# Add styles

Add some classes and styles, for instance:

<Page xmlns="http://schemas.nativescript.org/tns.xsd"

actionBarHidden="true"

loaded="onQuizListPageLoaded">

<GridLayout rows="auto, \*"

class="quizList-container">

<Label text="Choose Your Quiz"

class="quizList-headerText imageBackgroundContainer" />

<ListView items="{{ quizList }}" row="1">

<ListView.itemTemplate>

<GridLayout columns="auto, \*" class="quizList-itemContainer">

<Image src="{{ image ? image : 'res://unknownquizimage' }}"

class="quizList-image quizImage"

stretch="aspectFill"/>

<Label text="{{ name }}" col="1" class="quizList-text"/>

</GridLayout>

</ListView.itemTemplate>

</ListView>

</GridLayout>

</Page>

In the quiz-list-view.css file:

.quizList-container{

*background-color*: silver;

}

.quizList-image{

*width*: 70;

*height*: 70;

*border-width*: 1;

*border-color*: rgb(38, 30, 88);

}

.quizList-text{

*font-size*: 15;

*vertical-align*: center;

*margin-left*: 20;

*color*: white;

}

.quizList-itemContainer{

*padding*: 10;

*background-color*: rgb(116, 114, 114)

}

.quizList-headerText{

*font-size*: 25;

*padding*: 20;

*margin-bottom*: 2;

*text-align*: center;

*align-self*: stretch;

*background-image*: url("res://choosetopicbackground");

}

And add this to the app.css file (to be used again later):

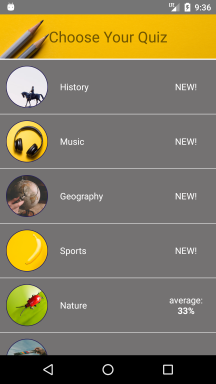
.quizImage{

*border-radius*: 50%;

*background-size*: cover;

}

You should be seeing something similar to this:



# A Small fix for extra items' appearance

You may notice that sometimes when hot-reloading you get extra items in your list. To avoid that, add a clearQuizzes function in the view model and call it each time before you load the quizzes:

In the view model (quiz-list-view-model.js):

viewModel.clearQuizzes = *function* () {

while (viewModel.length) {

viewModel.pop();

};

}

In the code-behind (quiz-list-view.js) paste this just before 'vm.loadQuizzes();':

vm.clearQuizzes();

# Add support for quiz average display

Some quizzes have as part of their data a 'gameCounter' property and an 'aggregatedScore' property. Putting aside the mock data, where the 'Nature' quiz holds these properties, they are meant to be added the first time a quiz is played and then updated on subsequent games. For quizzes who hold such data, you'll want to display what's the average grade users got when playing it.

##### Add Average Score Data to the View-Model

To have this data as part of the items in the view model, add a 'getAverageScore' function inside the view model, and change the 'addQuizzesToViewModel' function to use it:

getAverageScore = *function* (*quiz*) {

*var* averageScore = (quiz.gameCounter && quiz.gameCounter > 0) ?

quiz.aggregatedScore / quiz.gameCounter : undefined;

averageScore = utilities.convertFractionToPercentageString(averageScore);

return averageScore;

}

addQuizzesToViewModel = *function* (*quizzes*) {

for (i = 0; i < quizzes.length; i++) {

/\* added \*/*var* averageScore = getAverageScore(quizzes[i]);

viewModel.push({

id: quizzes[i].\_id,

name: quizzes[i].name,

image: quizzes[i].image,

/\* added \*/averageScore: averageScore,

/\* added \*/averageScoreExists: typeof averageScore !== 'undefined'

});

};

}

The 'getAverageScore' function uses a function that is found on a separate file – you'll need to require it:

*var* utilities = require("../../shared/utilities");

Now create this file in the 'shared' folder. Name it 'utilities.js' and paste in this:

*exports*.convertFractionToPercentageString = *function* (*number*) {

if (typeof number !== 'undefined') {

return `${Math.round(number \* 100)}%`;

}

return undefined;

}

##### Update the view with the average score data

Finally, make this data displayed as part of the itemTemplate of the quiz-list ListView:

<ListView.itemTemplate>

<GridLayout columns="auto, \*, 100" class="quizList-itemContainer">

<!-- added one column to GridLayout -->

<Image src="{{ image ? image : 'res://unknownquizimage' }}"

class="quizList-image quizImage"

stretch="aspectFill"/>

<Label text="{{ name }}"

col="1"

class="quizList-text"/>

<!-- added label:-->

<Label col="2"

class="quizList-text quizList-quizAverage"

textWrap="true"

visibility="{{ averageScoreExists? 'visible' : 'collapsed' }}">

<FormattedString>

<Span text="average:&#xa;"/>

<Span text="{{ averageScore }}"

fontWeight="Bold"/>

</FormattedString>

</Label>

<!-- added label:-->

<Label col="2"

class="quizList-text quizList-quizAverage"

visibility="{{ !averageScoreExists? 'visible' : 'collapsed' }}"

text="NEW!"/>

</GridLayout>

</ListView.itemTemplate>

Note:

* The visibility checks for 'averageScoreExists' are there to make sure that if average score does not exist for a quiz, 'NEW!' should be displayed instead.
* Using FormattedString inside labels allows making line breaks (if "textWrap" property is set to 'true') and to specify different formats for parts of the label.

An alternative would be to use a container for several labels, but that is slightly more costly performance-wise.

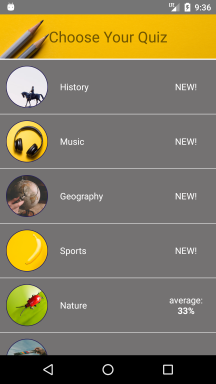
Also add this to the view's css file:

.quizList-quizAverage{

*text-align*: center;

}

This should be the result:



# Connect to Back-End data

##### Configure server access

Create a config.js file in your shared folder. From this file you'll export your API URL for the server and a parameter that states whether you'll be using server data or local mock data.

As your server is on your localhost, which is called in different ways by Android and iOS, some preparation for the API URL assignment is needed. Require the 'platformModule', which gives access to the type of operating system the device has:

*var* platformModule = require("platform");

Create a parameter to hold the different localhost access paths and assign it with a value according to the OS in use:

*var* localHostAccessPath;

if (platformModule.device.os === platformModule.platformNames.ios) {

localHostAccessPath = "localhost";

}

else if (platformModule.device.os === platformModule.platformNames.android) {

localHostAccessPath = "10.0.2.2";

}

Finally, export the two parameters:

*module*.*exports* = {

useLocalData: false,

apiUrl: "http://" + localHostAccessPath + ":3001/"

};

Set the 'userLocalData' parameter to false to be able to connect and utilize server data.

##### Utilize server data

Import the new module you've created to your quiz-list-view-model file:

*var* config = require("../../shared/config");

In the loadQuizzes function, check if quizzes should be loaded from local data or from the server and direct to the appropriate functions:

viewModel.loadQuizzes = *function* () {

if (config.useLocalData) {

loadMockDataQuizzes();

}

else{

loadBackEndDataQuizzes();

}

}

Paste the loadBackEndDataQuizzes function in the quiz-list-view-model file, aimed to get a collection of quizzes and then handle them the same way as before:

loadBackEndDataQuizzes = *function* () {

return fetch(config.apiUrl + 'quizzes')

.then(handleErrors)

.then(*function* (*response*) {

return response.json();

})

.then(*function* (*data*) {

addQuizzesToViewModel(data);

});

};

Also add this error handling function in the bottom of your view-model file:

handleErrors = *function* (*response*) {

if (!response.ok) {

*console*.log(JSON.stringify(response));

throw *Error*(response.statusText);

}

return response;

}

# Add a loading indicator

Now that you're getting real data from the server, you may also have a small lag between loading the view and having the list of quizzes loaded.

If you do not have a lag, or if you still want to work with local mock data (this will give you almost all functionality for this workshop, except affecting the average after completing a quiz), change your 'loadQuizzes' function in the view model by adding a timeout and returning a promise:

viewModel.loadQuizzes = *function* () {

if (config.useLocalData) {

return new Promise(*resolve* *=>*

setTimeout(resolve, 3000)

).then(loadMockDataQuizzes);

}

else {

return new Promise(*resolve* *=>*

setTimeout(resolve, 3000)

).then(loadBackEndDataQuizzes);

}

}

To indicate the users there's something in progress rather than have them think the app is stuck, add an activity indicator to the xml, as the last element in the GridLayout, just before the </GridLayout> declaration in the bottom:

<ActivityIndicator busy="{{ isLoading }}" row="1" />

Now you need to set the "isLoading" flag to be true when quizzes are loading. Do that in the quiz-list.js file, in the 'onQuizListPageLoaded' function, just after binding the quizListData to the page as its binding context:

quizListData.set("isLoading", true);

To turn the flag off when the quiz list finishes loading, you need to turn the "isLoading" flag to false. These lines should replace the ' vm.loadQuizzes ()' line, and they are the ones that expect the promise you've now added to the 'loadQuizzes' function:

vm.loadQuizzes().then(*function* () {

quizListData.set("isLoading", false);

});

# Apply styling according to operating system

In iOS, the indicator looks ok. But if you're using Android, you'll see the activity indicator takes up the whole screen and is thus quite ugly.

It'd be a good idea to add styling for this indicator, but just for android. You'll do that here by importing to the main .css file (app.css) another file, which the application will choose according to its suffix.

In the tools/assets folder, add two files: platformStyles.android.css and platformStyles.ios.css.

Import them to your app.css file with this line:

@import './tools/assets/platformStyles.css';

Add this style to the .android file:

ActivityIndicator{

*width*: 50;

*height*: 50;

*align-self*: center;

*vertical-align*: center;

}

And add some other style you'll be able to see, say this, to the .ios file:

.quizList-quizAverage{

*background-color*: rgb(98, 65, 84);

*border-radius*: 10;

}

The Nativescript compiler takes into account only files with a suffix relevant to the OS you're using. This is true for all files, when such a suffix exists, as well as to .css files as in this example.

# Add navigations

##### Create a navigation module

You'll be having a few navigations implemented in the app, which will be handled more easily through a 'navigation module'. Create it by creating a file named 'navigation.js' in the 'shared' folder.

Require the frame module, which navigation in NativeScript relies on:

*var* frameModule = require("ui/frame");

##### apply navigation from the welcome view to the quiz-list view

In navigation.js, create and export a function that will handle the navigation to the list view:

*exports*.goToQuizList = *function* () {

*var* topmost = frameModule.topmost();

topmost.navigate("views/quiz-list-view/quiz-list-view");

}

This function tells the topmost frame, the one the user sees, to navigate to the quiz-list view.

Go to welcome-view.js and at the top of the file require the new module:

*var* navigationModule = require("../../shared/navigation");

And now change the onButtonTapped event handler to this:

*exports*.onButtonTapped = *function* (*args*) {

navigationModule.goToQuizList();

}

You can also erase the 'onPageLoaded' function while you're there, it was just for demonstration purposes.

##### Change the starting view back to the welcome-view:

The navigation module is a good place to hold control over the start-up view. To support that, paste the following function in navigation.js:

*exports*.startupView = *function* () {

return "views/welcome-view/welcome-view";

}

Import the navigation module in the app.js file:

*var* navigationModule = require("./shared/navigation");

Replace the moduleName for the application.start() definition with this:

application.start({ moduleName: navigationModule.startupView() });

Make sure all is working as expected, and you manage to navigate to the quiz-list from the welcome view. Then, feel free to change the startup view to be the quiz-list view for now (it'd now be easier to do from the navigation.js file).

# Handle Quiz Selection

##### add a handler to the quiz-list item tap

To choose a quiz, the user needs to tap on an item in the list. Add an eventHandler for this on the ListView in the quiz-list-view.xml file and hook it to the 'itemTap' event:

itemTap="onSelectQuiz"

In the code-behind quiz-list, add a temporary handler, just to see you're getting the right data upon item-tapping:

*exports*.onSelectQuiz = *function* (*args*) {

*var* selectedQuizData = args.view.bindingContext;

*console*.log('selected quiz name: ' + selectedQuizData.name);

}

##### Create a quiz view-model

To handle this data and to start a new quiz using it, use a new view-model: in the view-models folder of your app, create a file named 'quiz-view-model.js'. You'll use this view model to access the questions data and make it accessible to the next views you'll create, and to post quiz scores when quizzes are finished.

Start by requiring the following modules in the top of the file:

*var* observableModule = require("data/observable");

*var* mockQuestionsData = require("../../mockData/mockQuestionsData.json");

*var* config = require("../../shared/config");

Declare the view model as an observable, utilizing the observable constructor to initialize it with data that you pass in when creating a new quizViewModel:

*function* quizViewModel(*quiz*) {

*var* viewModel = new observableModule.fromObject(quiz)

viewModel.currentScore = 0;

//paste viewModel functions here

return viewModel;

}

*module*.*exports* = quizViewModel;

The new quizViewModel will have upon its creation the same data as the one held in the quiz list for it plus an extra piece of data you're adding here: the users' current score in the quiz, which is always zero, as they have yet to answer any questions.

##### Add related questions to the quiz view-model

Next, add functions that will use the quiz's id to load related questions to it.

As with quiz-list loading, this will be done either from the local data or from the server, according to what's in your config.js file, and will return a promise. When loading quizzes from the back-end, the server already handles the questions' filtering for us, but for local data you need to do this yourself:

loadBackEndDataQuestions = *function* () {

return fetch(config.apiUrl + 'quizzes/' + quiz.id + '/questions')

.then(handleErrors)

.then(*function* (*response*) {

return response.json();

})

.then(*function* (*data*) {

viewModel.questions = data;

})

};

loadMockDataQuestions = *function* () {

*var* quizRelatedQuestions = mockQuestionsData.filter(*q* *=>* q.quiz\_id === quiz.id)

viewModel.questions = quizRelatedQuestions;

}

viewModel.loadQuestions = *function* () {

if (config.useLocalData) {

return new Promise(*resolve* *=>*

setTimeout(resolve, 2000)

).then(loadMockDataQuestions);

}

else {

return new Promise(*resolve* *=>*

setTimeout(resolve, 2000)

).then(loadBackEndDataQuestions);

}

}

##### Connect the quiz view-model to the code-behind

In your quiz-list-view.js, require the QuizViewModel:

*var* QuizViewModel = require('../../shared/view-models/quiz-view-model');

Replace the recently created 'onSelectQuiz' function with the following function:

*exports*.onSelectQuiz = *function* (*args*) {

*var* selectedQuizData = args.view.bindingContext;

*var* quiz = new QuizViewModel(selectedQuizData);

quizListData.set("isLoading", true);

quiz.loadQuestions().then(*function* () {

quizListData.set("isLoading", false);

*var* quizLength = quiz.questions.length;

*console*.log('quizLength: ' + quizLength);

});

}

Here, you're initializing a new QuizViewModel using the binding context of the selected quiz. You are then loading questions to it. As previously done for the loading of the list of quizzes, the data context's "isLoading" property is set to true while this is going on.

The quizLength variable will be used soon, for now it can give a good indication if all is in order so far (check, for example, that you're getting zero only for the 'Movies' quiz);

Next you'll be using the data for this quiz to navigate between its questions, but to do that you'll first need to create a question view and a question view-model, and to support navigation with context transfer to this new view.

Create The question view

# Initialize the question-view

Create a new folder in the 'views' folder named 'question-view'. Create three files in it with the same name and with the suffixes .js, .xml and .css.

In the .xml file, paste the following initial markup:

<Page xmlns="http://schemas.nativescript.org/tns.xsd"

actionBarHidden="true"

navigatingTo="questionPageNavigatingTo"

loaded="onQuestionPageLoaded">

<StackLayout>

<Label text="you've reached the question view!"/>

<Label text="{{'question id: ' + question.\_id}}"/>

</StackLayout>

</Page>

Note that this markup adds two eventHandlers to the page: one that hooks up to its 'loaded' event, as before, and one that hooks-up to its 'navigatingTo' event, which will give you access navigation data to the page.

# Add navigation to the question-view

Use the navigation module to navigate to this page by adding the following function to navigation.js:

*exports*.goToQuestionView = *function* () {

*var* topmost = frameModule.topmost();

topmost.navigate({

moduleName: 'views/question-view/question-view',

});

}

Notice that the syntax used here is different than the one used before for navigation calls: the moduleName is specified as part of an object. The reason for that is so that other properties can be added to this object, to support data transfer.

In the quiz-list-view.js file, require the navigation module:

*var* navigationModule = require("../../shared/navigation");

Call the recently added 'goToQuestionView' function at the end of the 'onSelectQuiz' function:

if (quizLength > 0) {

navigationModule.goToQuestionView();

}

Make sure navigation to the 'question-view' is working. It should work for all quizzes but the 'Movies' quiz.

##### Handle an empty quiz using the dialogs module

Utilize the dialogs module to handle the cases where the user chooses a quiz that has no questions related to it (quizLength > 0 is false):

First – require it –

*var* dialogs = require("ui/dialogs");

Next- add an 'else' after the previous check:

if (quizLength > 0) {

navigationModule.goToQuestionView();

}

else {

dialogs.alert("No questions were found for this quiz. Please try another one.").

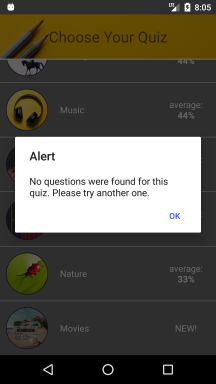
then(*function* () {

*console*.log("error loading questions. notification dialog closed.");

});

}

You can test this is working by tapping the 'Movies' quiz. This should be the result:



##### Expand navigation to the question view

To be able to navigate to specific questions from the quizzes, you'll need to pass the quiz data to the navigation function. This will be done in the 'onSelectQuiz' handler in quiz-list.js like this:

navigationModule.goToQuestionView(quiz, 0);

You're also passing a zero there, as this is the index of the first quiz question.

The 'goToQuestionView' function can now take this into account, and pass these on as the context of the navigation data:

*exports*.goToQuestionView = *function* (*quiz*, *index*) {

*var* topmost = frameModule.topmost();

topmost.navigate({

moduleName: 'views/question-view/question-view',

context: { quiz: quiz, currentQuestionIndex: index },

backstackVisible: false,

transition: {

name: "slide",

duration: 350,

curve: "easeIn"

}

});

}

Note that in the navigate command you're also setting the 'backstackVisible' property of the question-view to false. This means that a user who navigated away from the question view will not be able to return to it by pressing back.

Also, by adding the 'transition' property, you're creating an animation every time you navigate to this page: this one is doing a 'slide' animation (by default it's from left to right) that lasts 350ms and uses an 'easeIn' curve.

##### Create a question view-model

To hold the logic for the question-view, create a view-model in the shared/view-models folder and name it 'question-view-model.js'. Again, use the observable module which helps keep track of changes in the UI and initialize the view model with the following:

*var* observableModule = require("data/observable");

*function* questionViewModel(*question*) {

*var* viewModel = new observableModule.fromObject(question);

//paste viewModel functions here

return viewModel;

}

*module*.*exports* = questionViewModel;

##### connect the view with the view-model

Paste the following in the question-view.js file. This adds the handlers for the 'loaded' event and the 'navigatingTo' event to the page, with the appropriate binding context object (here, 'questionData', another observable wrapper), necessary imports, and some global variables you'll be using soon:

*var* QuestionViewModel = require('../../shared/view-models/question-view-model');

*var* observableModule = require("data/observable");

*var* vm;

*var* quiz

*var* questionIndex;

*var* quizLength;

*var* questionData = new observableModule.Observable();

*exports*.onQuestionPageLoaded = *function* (*args*) {

*var* page = args.object;

page.bindingContext = questionData;

}

*exports*.questionPageNavigatingTo = *function* (*args*) {

*var* page = args.object;

*var* context = page.navigationContext;

quiz = context.quiz;

questionIndex = context.currentQuestionIndex;

quizLength = quiz.questions.length;

vm = new QuestionViewModel(quiz.questions[questionIndex]);

questionData.question = vm;

}

Here, the handler for the navigatingTo event gets the context from the object passed during navigation and uses it to create a new instance of the QuestionViewModel, instantiating it with the question with the relevant index out of the chosen quiz's questions. It also sets this instance as part of the 'questionData' observable object.

The handler for the loaded event then sets the binding context for the page as this 'questionData' object.

Setting the pages binding context is also possible through the navigatingTo handler, but this causes unwanted behavior of the page displaying its content before an animated transition is finished.

Your view should now be able to display the little of the question you've set it to show:



# Expand the question-view

##### Add a progress property:

To indicate the user their progress in the quiz, add a progress property to the page's binding context, by adding this line in the 'questionPageNavigatingTo' event handler, after setting the questionIndex and the quizLength:

questionData.progress = `${questionIndex + 1} / ${quizLength}`;

##### Re-arrange the question-view to display The question's data

Replace the StackLayout in your question-view.xml file with this GridLayout, which already holds the correct bindings to your questionData properties and to those of the QuestionViewModel inside these:

<GridLayout rows="auto, auto, auto,\*"

class="questionContainer">

<Label text="{{progress}}"

row="0"

class="progressText"/>

<Label text="{{question.questionText}}"

row="1"

class="questionText"/>

<Image src="{{question.questionImage}}"

row="2"

class="questionImage"

visibility="{{question.questionImage ? 'visible' : 'collapsed'}}"

horizontalAlignment="center"

stretch="none"/>

<ListView items="{{ question.answers }}"

row="3"

separatorColor="transparent"

verticalAlignment="bottom">

</ListView>

</GridLayout>

You should now see something similar to this in your question views (for all quizzes apart from the Geography quiz – the questions there are a different case which will be handled later):



Note that the Geography quiz's questions are built differently and so are not displayed in this way. Support for this quiz's questions will be added at the last part of this workshop.

##### Add some color:

To make the application come alive a little, have the background color for the question-view be chosen randomly from a list of colors.

Add the following function in the code-behind question-view.js file:

setBackgroundColor = *function* () {

*var* colors = ['#58406D', '#314D70', '#E54B04', '#007461', '#655672', '#6B0758', '#513EE1', '#E00481', '#4D989E', '#3F7F47'];

*var* backgroundColor = colors[Math.floor(Math.random() \* colors.length)];

questionData.backgroundColor = backgroundColor;

}

Call it from the 'questionPageNavigatingTo' event handler with this:

setBackgroundColor();

And add this to either the page or the GridLayout in the .xml file:

backgroundColor="{{backgroundColor}}"

##### Add styles

Paste the following in the question-view.css file:

.progressText{

*text-align*: center;

*color*: white

}

.questionText{

*text-align*: center;

*margin-top*: 40;

*margin-bottom*: 20;

*font-size*: 25;

*white-space*: normal;

*color*: white

}

.questionContainer{

*padding*: 10, 20;

}

.questionImage{

*max-height*: 150;

*border-width*: 3;

*border-color*: black;

*border-radius*: 10;

}

You should see something like this:



Don't worry about how the answers look for now: soon you'll be changing the 'answers' property on the QuestionViewModel, and you'll add an ItemTemplate to the ListView accordingly.

#### Tip: styling items in non-object arrays

If you were to remain with the original 'answers', which is currently an array of strings, you could still specify an ItemTemplate for your items. This would allow you to apply styling to any component in the template.

To do that, you'd need to use the following binding, which binds to each item's string value, and add class assignments where you like:

<ListView.itemTemplate>

<Label text="{{ $value }}" />

</ListView.itemTemplate>

# Add functionality – Give the User feedback When choosing an answer

So far in the question view, you've just presented the question and possible answers. Next, you'll want the user to be able to tap an answer, get feedback if they were right or wrong and then navigate to the next question page.

To start, in question-view.xml, attach an itemTap handler to the answers' ListView:

itemTap="onSelectMultipleChoiceAnswer"

In question-view.js, add this function:

*exports*.onSelectMultipleChoiceAnswer = *function* (*args*) {

*var* chosenAnswer = args.view.bindingContext;

*var* answeredCorrectly = vm.checkMultipleChoiceAnswer(chosenAnswer);

*console*.log('answeredCorrectly: ' + answeredCorrectly);

}

##### Add functions to the question view-model

To be able to signal the user that the answer they have tapped is correct or not, and to mark the correct answer if not, re-create the answers array in the QuestionViewModel.

Add a function called ' initQuestion' in the question-view-model.js file (above the 'return viewModel;' line):

viewModel.initQuestion = *function* () {

*var* newAnswersArray = [];

if (question.answers) {

for (i = 0; i < question.answers.length; i++) {

*var* answer = {

answerText: question.answers[i],

isSelected: false,

showCorrect: false,

isCorrect: i === question.correctAnswerIndex

};

newAnswersArray.push(answer);

}

viewModel.answers = newAnswersArray;

}

}

This function changes the "answers" property on the view model, if such property should exist. Instead of holding an array of strings, it now holds an array of objects, with the previous string (now 'answerText') just one of them.

Call the 'initQuestion' function from the question-view.js file in the 'questionPageNavigatingTo' handler, just after assigning 'vm' to be a new QuestionViewModel:

vm.initQuestion();

Going back to the question view-model, also add the following function (also above the 'return viewModel;' line). This is the function you're calling from the itemTap handler:

viewModel.checkMultipleChoiceAnswer = *function* (*chosenAnswer*) {

chosenAnswer.isSelected = true;

*var* answers = viewModel.get("answers");

*var* correctAnswer = answers[question.correctAnswerIndex];

correctAnswer.showCorrect = true;

return chosenAnswer.isCorrect;

};

This function first sets the 'isSelected' property of the chosen answer to true, and then sets the 'showCorrect' property of the correct answer to true.

##### Add an Answer item template

An itemTemplate on the answers ListView in question-view.xml needs to be added now, otherwise each item will be represented in the view as the text [object Object]:

<ListView.itemTemplate>

<ContentView>

<GridLayout columns="\*, auto"

class="{{showCorrect ? 'answerItemContainer answerItemContainerShowCorrect' : 'answerItemContainer'}}">

<Label text="{{ answerText }}"

class="{{(isSelected && !isCorrect) ? 'answerItemText answerItemTextSelectedWrong' : 'answerItemText'}}"/>

<Image col="1"

visibility="{{isSelected ? 'visible' : 'hidden'}}"

class="resultIndicator" src="{{isCorrect? 'res://vmark' : 'res://xmark'}}"

stretch="aspectFill"/>

</GridLayout>

</ContentView>

</ListView.itemTemplate>

This template presents the 'answerText' string property you've added to the new answer object, and utilizes the 'showCorrect', 'isCorrect' and 'isSelected' boolean properties to determine class assignments, visibility and an image source, all to support feedback to the user.

Note the 'ContentView' wrapping the GridLayout: a ContentView is a view that can hold a single child and serves as its placeholder. It seems redundant here, but if you try to remove it you'll find that (at least in android) the margin between the items will not work.

Add some styling:

.answerItemText{

*font-size*: 15;

*color*: black;

}

.answerItemTextSelectedWrong{

*color*: red;

}

.answerItemContainer{

*padding*: 15, 10;

*margin-top*: 10;

*border-width*: 2;

*border-color*: rgb(43, 53, 50);

*border-radius*: 10;

*background-color*: white;

}

.answerItemContainerShowCorrect{

*border-width*: 5;

*padding*: 12,7;

*border-color*: limegreen;

*animation-name*: correctAnswerColorAnimation;

*animation-duration*: 1s;

*animation-fill-mode*: forwards;

}

@keyframes *correctAnswerColorAnimation* {

0% {

*background-color*: white;

}

100% {

*background-color*: rgb(112, 241, 112);

}

}

.resultIndicator{

*width*: 20;

*height*: 20;

}

Try selecting answers from the answer list. You should get correct logging telling you if your answer is true or false, but still no visual representation on the app.

##### Convert the answers array to an array of observables

The reason for not getting visual feedback is that each of the answers is a normal object, and does not properly notify its changes to the UI. To correct this, make each of the answers an Observable, by creating them using the observableModule. In the question-view-model.js file, replace the var answer creation in the 'initQuestion' function with this:

*var* answer = new observableModule.fromObject({

answerText: question.answers[i],

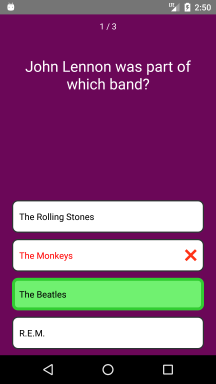
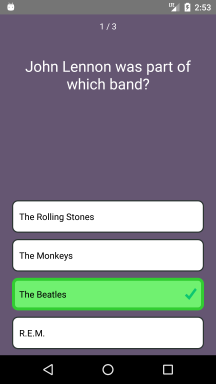
isSelected: false,

showCorrect: false,

isCorrect: i === question.correctAnswerIndex

});

This should have any tap a user taps on an answer give a visual indication of its correctness by showing an image of a v mark or an x mark, and also mark the correct answer's background and border:

# Additional functionality: loose ends

##### Disable Item Taps registering after choosing an answer

At the moment, any tap of an answer item can be followed by other taps on different answers, resulting in conflicting visual representations.

To avoid this, add the following function to question-view.js, which removes the itemTap event listener from the list:

disableList = *function* (*args*) {

*var* parent = args.object.parent;

if (parent) {

*var* list = view.getViewById(parent, "answersListView");

if (list) {

list.removeEventListener(listViewModule.ListView.itemTapEvent);

}

}

}

You'll need to require the following to get the 'view' module and the listViewModule usable:

*var* view = require("ui/core/view");

*var* listViewModule = require("ui/list-view");

Also, add the relevant id on the ListView in the .xml file:

id="answersListView"

Finally, add a call to the function at the beginning of the itemTap handler function (the 'onSelectMultipleChoiceAnswer' function):

disableList(args);

##### Increment the user's score in the quiz

Add the 'incrementScore' function in the quiz-view-model.js file:

viewModel.incrementScore = *function* () {

*var* newScore = viewModel.currentScore + 1;

viewModel.currentScore = newScore;

}

Add a call to this function on the 'onSelectMultipleChoiceAnswer', after getting the result telling you if the user was right or wrong:

if (answeredCorrectly) {

quiz.incrementScore();

}

##### Navigate to next question page

In question-view.js, Require the navigationModule:

*var* navigationModule = require("../../shared/navigation");

Add the 'navigateToNextPage' function, which uses the navigationModule:

navigateToNextPage = *function* () {

if (quizLength > questionIndex + 1) {

navigationModule.goToQuestionView(quiz, questionIndex + 1);

}

else {

*console*.log('quiz ended');

};

}

This function checks if there are still questions left on the quiz. If so, it navigates to the next question by passing as context the quiz and an incremented index. If the user has reached the end of the quiz, it'll just log this to the console for now.

Add a delayed call to this function at the end of the 'onSelectMultipleChoiceAnswer' function:

setTimeout(navigateToNextPage, 1000);

This gives the user one second to see whether they were correct or not (and if not see the right answer) before the app will automatically navigate to the next question.

##### Finalize quiz Score Upon Quiz End

In the quiz-view-model.js file, require the utilities module:

*var* utilities = require("../../shared/utilities");

Add these functions to the viewModel:

postScore = *function* (*finalScore*) {

return fetch(config.apiUrl + 'quizzes/' + quiz.id, {

method: "PATCH",

body: JSON.stringify({

score: finalScore,

}),

headers: {

"Content-Type": "application/json"

}

}).then(handleErrors);

}

viewModel.finalizeScore = *function* () {

*var* correctAnswers = viewModel.currentScore;

*var* quizLength = viewModel.questions.length;

*var* finalScore = correctAnswers / quizLength;

viewModel.set("finalScore", finalScore);

*var* presentableScore = utilities.convertFractionToPercentageString(finalScore);

viewModel.set("presentableScore", presentableScore);

if (!config.useLocalData) {

postScore(finalScore);

}

};

The 'finalizeScore' function determines what the user's final score is, and adds this as a property to the current QuizViewModel. It also adds a property you'll use later, that already holds the same score but ready for presentation.

If the app is connected to a server, it also posts this score to the server.

This should be done when a quiz is ended, so add a call to the 'finalizeScore' function at the 'else' section of the 'navigateToNextPage' function in question-view.js:

quiz.finalizeScore();

Create a quiz summary view

# Initialize the quiz summary view

In the views folder, create a new folder and name it 'quiz-summary-view'. Create three files with the same name there, with the .js, .xml and .css suffixes.

Paste this into the quiz-summary-view.xml file, to display existing data for the quiz – its image, name and its (previous) average, if it exists:

<Page xmlns="http://schemas.nativescript.org/tns.xsd"

actionBarHidden="true"

loaded="summaryViewLoaded">

<GridLayout rows="auto, auto, auto, auto, \*, auto, auto"

class="imageBackgroundContainer summary-gridContainer">

<Image src="{{image ? image : '~/images/unknownImage.png'}}"

class="quizSummaryImage quizImage"

stretch="aspectFill"

row="0"

rowSpan="2"

horizontalAlignment="right"/>

<Label text="{{name}}"

row="0"

class="quizName"

horizontalAlignment="left"/>

<StackLayout row="2"

horizontalAlignment="center">

<Label text="Your Score:"

class="scoreTitleText" />

<Label text="{{presentableScore}}"

class="scoreText userScoreText"/>

</StackLayout>

<StackLayout row="3"

class="communityScoreContainer"

horizontalAlignment="center"

orientation="horizontal"

visibility="{{averageScoreExists? 'visible' : 'collapsed'}}">

<Label text="Community Average:"

class="scoreTitleText" />

<Label text="{{averageScore}}"

class="scoreText averageScoreText" />

</StackLayout>

</GridLayout>

</Page>

Add styling in the .css file:

.quizSummaryImage{

*width*: 250;

*height*: 250;

*border-width*: 3;

*border-color*: rgb(3, 81, 83);

*border-radius*: 125;

*margin-bottom*: 20;

}

.summary-gridContainer{

*padding*: 25;

*background-image*: url("res://summaryviewbackground");

}

.quizName{

*border-radius*: 50%;

*background-color*: rgba(256,256,256,0.65);

*background-color*: black;

*height*: 50;

*padding-right*: 15;

*padding-left*: 15;

*min-width*: 180;

*font-size*: 22;

*color*: white;

*text-align*: center;

*vertical-align*: center;

*white-space*: normal;

}

.scoreTitleText{

*font-size*: 22;

*vertical-align*: center;

*text-align*: center;

}

.scoreText{

*vertical-align*: center;

*text-align*: center;

}

.userScoreText{

*color*: black;

*font-size*: 44;

}

.averageScoreText{

*margin-left*: 5;

*font-size*: 22;

}

.communityScoreContainer{

*margin-top*: 20;

}

# Add navigation to the quiz summary view

In the navigation.js file, add a function that will navigate the app to this view. This Time, you'll be setting the view's dataContext during navigation:

*exports*.goToQuizSummaryView = *function* (*quiz*) {

*var* topmost = frameModule.topmost();

topmost.navigate({

moduleName: 'views/quiz-summary-view/quiz-summary-view',

bindingContext: quiz,

backstackVisible: false,

transition: {

name: "slideTop",

duration: 350,

curve: "easeIn"

}

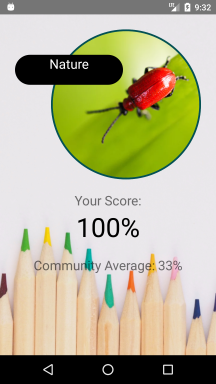
});

}

Use this function in the question-view.js file, by calling it upon quiz-end in the 'navigateToNextPage' function, just after the 'quiz.finalize();' call:

navigationModule.goToQuizSummaryView(quiz);

If you play a quiz all the way through, you should now see something like this:



# Correct alignment in android

As apparent in the above screenshot, Android users will see that the quiz-name text is not vertically aligned to the center of its container (the container also being part of the label itself, applied in the style for it). For iOS users it'll be properly vertically aligned to center, as in iOS this is the default vertical alignment. You can fix this directly through the Android Gravity class, by hooking up to the loaded event of this label.

Add a 'loaded' eventHandler to the label binded to "{{quiz.name}}" in the .xml file:

loaded="quizNameLoaded"

In the code behind file, add the following handler:

*exports*.quizNameLoaded = *function* (*args*) {

if (args.object.android) {

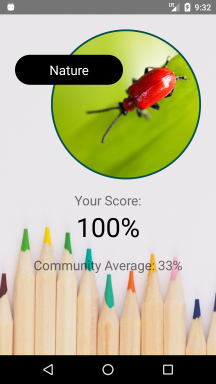
args.object.android.setGravity(17);

}

}

17 is the android gravity code for placing an object in the center of its container in both the vertical and horizontal axis.

You should then see the text properly centrally aligned:



# Add a share option:

Naturally, after completing a quiz, the app's users will want to brag to their friends about their achievements. To allow that, you'll now hook the app to a social-share plugin that will find the devices share options and use them to share a text with others.

##### Install the nativescript-social-share plugin

Install the plugin using the following command in your command-line (when you're at the root folder of your app):

tns plugin add nativescript-social-share



##### Add a share button

Add a share icon to the quiz-summary-view.xml, inside the GridLayout. It is placed in a ContentView to help with its alignment, and it calls to the image the 'share' image resource:

<ContentView class="shareIconContainer"

row="2"

horizontalAlignment="right"

tap="shareTapped">

<Image src='res://share'

stretch="AspectFill"

horizontalAlignment="center"

verticalAlignment="center"/>

</ContentView>

Add this style to the .css file accordingly:

.shareIconContainer{

*width*: 50;

*height*: 50;

*border-radius*: 25;

*background-color*: rgba(3,81,83, 0.8);

*vertical-align*: bottom;

*margin-right*: 30;

}

##### Apply share functionality

In the quiz-summary-view.js file, require the newly installed plugin:

*var* socialShare = require("nativescript-social-share");

Get a reference to the view's dataContext through the 'loaded' event of the page, by creating a variable named 'vm' and adding the 'summaryViewLoaded' handler:

*var* vm;

*exports*.summaryViewLoaded = *function* (*args*) {

*var* page = args.object;

vm = page.bindingContext;

}

Add the following handler for the tap event of the icon:

*exports*.shareTapped = *function* (*args*) {

*var* userScore = vm.get("presentableScore");

*var* quizName = vm.get("name");

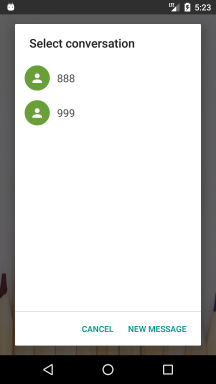
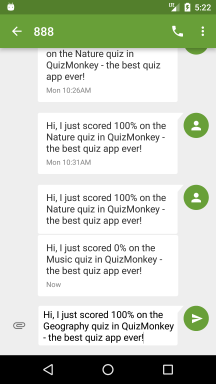
socialShare.shareText("Hi, I just scored " + userScore + " on the " + quizName + " quiz in QuizMonkey - the best quiz app ever!");

}

This function extracts the user's score in the quiz and the quiz name, and puts them as part of a text to be socially shared.

To see this working, you may need to run tns run android/ios again in the command line, as hot-reloading may have stopped working with the new installation. If you get errors in the build process delete the folder relevant to the OS you're using (Android/iOS) in your 'platforms' folder and try running again.

Now, when you tap the share image, you'll get to choose with which app you'll want to share your 'shareText' and continue to sharing using it.

# wrap-up: navigate back to the quiz-list

Finally, here, you'll just add a button that will navigate the user back to the quiz-list. In it the quizzes will be re-loaded and if the app is connected to a server the user will be able to see an updated average score for the quiz they just took.

In the quiz-summary-view.js file, require the navigation module:

*var* navigationModule = require("../../shared/navigation");

Add this tap event handler:

*exports*.backToQuizListTapped = *function* (*args*) {

navigationModule.goToQuizList();

}

Now add a button with this handler listed as its tap event to the .xml file, inside the GridLayout:

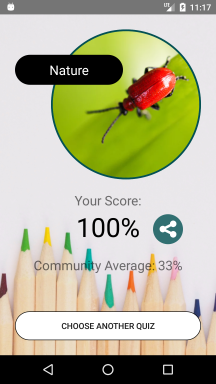
<Button text="choose another quiz"

tap="backToQuizListTapped"

class="navigationButton"

row="5"/>

By tapping on the new button, you can now safely navigate back to the quiz-list at the end of each quiz:



Question-view enhancements

**Note: The two parts in this section are an addition to the main flow of the app. They can be executed before adding the 'quiz-summary-view', but they make take a while to complete.**

# add a 'leave quiz' button and handle Android's back event

##### Add a 'leave quiz' button

Here, to allow the user to change their mind and leave a quiz in the middle, you'll add a 'leave quiz' label. Tap events can be added to any element, and with this one, upon tapping, a confirmation dialog will open, verify this is what the user wants, and if so take them back to the quiz-list.

Add a label with a tap event to the question-view's main GridLayout.

<Label text="Leave Quiz"

row="0"

class="progressText"

tap="onLeaveTapped"

horizontalAlignment="left"/>

In question-view.js, require the dialog module:

*var* dialogs = require("ui/dialogs");

Add the tap event handler, which utilizes a 'confirm' dialog and in this case takes on an object:

*exports*.onLeaveTapped = *function* () {

dialogs.confirm({

title: "Back/Leave Button Pressed",

message: "Leaving now will take you the quiz-list page and erase this quiz's progress",

okButtonText: "Leave Quiz",

cancelButtonText: "Cancel",

}).then(*function* (*result*) {

if (result) {

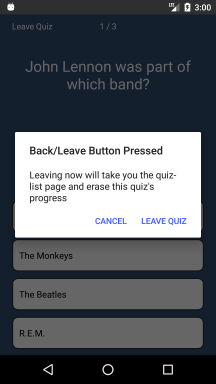
navigationModule.goToQuizList();

}

});

}

The result of the confirm dialog is a Boolean of true (user says ok) or false (user cancelled). Here, if the user chooses cancel, nothing happens, and if they choose ok they are taken to the quiz-list-view.



##### Handle Android's back event pressed

On Android devices, the user can press the back button at any point and the app will automatically take them back. In the function that navigates the user to the question-view, in the navigation module, you've set the question-view to be 'forgotten' by the backstack, so from any question-view it'll navigate back to the quiz-list.

This could be upsetting to a user who pressed this button accidentally, in the middle of a quiz, or if they pressed it without realizing the results. For this reason, here, you'll hook the back-pressed event so that it has the same functionality as pressing the 'Leave Quiz' label:

In question-view.js, require the following module:

*var* application = require('application');

Add this to the 'onQuestionPageLoaded' handler:

if (application.android) {

application.android.addEventListener(application.AndroidApplication.activityBackPressedEvent, backOrLeavePressed);

}

Add an 'onQuestionPageUnloaded' handler, needed so that pressing back in other pages will not trigger the same ' backOrLeavePressed' function from here:

*exports*.onQuestionPageUnloaded = *function* () {

if (application.android) {

application.android.removeEventListener(application.AndroidApplication.activityBackPressedEvent, backOrLeavePressed);

}

};

And attach it to the 'unloaded' event of the page in the .xml file:

unloaded="onQuestionPageUnloaded"

Now, add this function 'backOrLeavePressed', very similar to the one you used before for the 'Leave Quiz' tap handler, apart from the condition in the beginning:

backOrLeavePressed = *function* (*args*) {

if (args) {

args.cancel = true;

}

dialogs.confirm({

title: "Back/Leave Button Pressed",

message: "Leaving now will take you the quiz-list page and erase this quiz's progress",

okButtonText: "Leave Quiz",

cancelButtonText: "Cancel",

}).then(*function* (*result*) {

if (result) {

navigationModule.goToQuizList();

}

});

}

By setting 'args.cancel = true', this function first cancels the automatic back-navigation event, if called from the back button. It then continues to present the same confirmation dialog, and to take the user to the quiz-list view if they wish do to so.

You can now change the 'onLeaveTapped' handler to call this new function:

*exports*.onLeaveTapped = *function* () {

backOrLeavePressed();

}

You should have both the 'Leave Quiz' button working, as well as the back button on an Android device, and they should both lead you to the same confirmation dialog with the same subsequent flow.

# Support the 'geography' quiz questions

Questions that are part of the geography quiz are not multiple choice questions. They are planned for google-maps integration, where a user is shown a map and is asked to pin a location on it. They are supposed to get feedback whether they managed to pin the location close enough to the correct location according to the question's data.

##### Install the Google Maps plugin

The first thing you'll need to do is install the [nativescript-google-maps-sdk](https://www.npmjs.com/package/nativescript-google-maps-sdk). Do this with this command in your app's root folder:

tns plugin add nativescript-google-maps-sdk



After installation, add the Google Maps API keys for each OS:

#### Android

Get a Google Maps API key for android here:

<https://developers.google.com/maps/documentation/android-api/signup>

Copy the file 'nativescript\_google\_maps\_api.xml' in this location:

node\_modules/nativescript-google-maps-sdk/platforms/android/res/values

To this location:

app/App\_Resources/Android/values

Uncomment the comment in the file and paste your API Key in the indicated place, where is says PUT\_API\_KEY\_HERE.

This is how the file should look:

<?xml version="1.0" encoding="utf-8"?>

<resources>

<string name="nativescript\_google\_maps\_api\_key">AIzaSyA8BO7Cgn7es7wb9qDRozwq15bFNm6gCeM</string>

</resources>

#### iOS

Get a Google Maps API key for iOS here:

<https://developers.google.com/maps/documentation/ios-sdk/get-api-key>

Add these lines to the app.js file (before application.start()):

if (application.ios) {

GMSServices.provideAPIKey("PUT\_API\_KEY\_HERE");

// GMSServices.provideAPIKey("AIzaSyBjhCQoXVLt2ByhXr2iQMgLo7iO0VJ8Y0k");

}

Replace the "PUT\_API\_KEY\_HERE" with your API key.

To see this working, run tns run android/ios again. It may work without it, but in general hot-reloading is less dependable after installing new dependencies. Also, you may get errors in the build process. To solve them, delete the folder relevant to the OS you're using (Android/iOS) in your 'platforms' folder and try tns run andoird/ios again.

Make sure your app is still running correctly before proceeding.

##### Display the map-view in the question-view

Add the google maps namespace to the question-view.xml Page:

xmlns:maps="nativescript-google-maps-sdk"

Add a mapView inside the main GridLayout:

<maps:mapView visibility="{{question.answers ? 'collapsed' : 'visible'}}"

row="3"

latitude="{{question.mapSettings.latitude}}"

longitude="{{question.mapSettings.longitude}}"

zoom="{{question.mapSettings.zoom}}"

mapReady="onMapReady"

coordinateTapped="onCoordinateTapped"/>

The mapView should be visible only when there is no collection of answers – which in our data can mean it only a 'map' question.

Note: the map will be loaded regardless of its visibility status. It may be better optimized to handle its loading conditionally, only for questions who need it, but for the performance needs of this tutorial this should be acceptable.

While you're here, you can also add to the ListView this visibility setting – the opposite to the one of the map:

visibility="{{question.answers ? 'visible' : 'collapsed'}}"

The mapView takes on here the latitude, longitude and zoom parameters from the question's data, setting it with its center point and its zoom level. It has two event handlers: one that hooks up to its 'mapReady' event, after it loads, and one that hooks up to the 'coordinateTapped' event, for when a user taps an area in the map.

##### Add functionality to the map

In the question-view.js, require the mapsModules, and add a variable which will hold a reference to the map view:

*var* mapsModule = require("nativescript-google-maps-sdk");

*var* mapView = null;

Add the 'onMapReady' handler, which will hook up the map to the added variable and add some extra settings to the map. Here, you'll be blocking almost all functionality of changing view angles and of scrolling the map, just to keep things a bit simpler:

*exports*.onMapReady = *function* (*args*) {

mapView = args.object;

mapView.settings.compassEnabled = false;

mapView.settings.indoorLevelPickerEnabled = false;

mapView.settings.mapToolbarEnabled = false;

mapView.settings.myLocationButtonEnabled = false;

mapView.settings.rotateGesturesEnabled = false;

mapView.settings.scrollGesturesEnabled = false;

mapView.settings.tiltGesturesEnabled = false;

mapView.settings.zoomControlsEnabled = false;

mapView.settings.zoomGesturesEnabled = false;

}

Add the 'onCoordinateTapped' handler:

*exports*.onCoordinateTapped = *function* (*args*) {

mapView.removeEventListener(mapsModule.MapView.coordinateTappedEvent);

*var* answeredCorrectly = vm.checkMapLocationAnswer(args.position.latitude, args.position.longitude);

showUserMarker(args, answeredCorrectly);

setTimeout(showCorrectMarker, 500);

if (answeredCorrectly) {

quiz.incrementScore();

}

setTimeout(navigateToNextPage, 2000);

}

This handler first removes the eventListener for tapping coordinates again, same as you did upon itemTap for the answers' ListView. It then turns to the questionViewModel to check if the tapped coordinate is correct. After getting an answer, it calls two functions that will show the tapped coordinate with a right/wrong indication and also show the correct location, with a small delay. If the user answers correctly, this function also calls a function that will increment their quiz score. Finally, it navigates to the next page, with a small delay, using the existing 'navigateToNextPage' function.

To use this handler, add these two functions to the code-behind:

showCorrectMarker = *function* () {

*var* correctMarker = new mapsModule.Marker();

*var* correctLat = vm.locationAnswer.latitude;

*var* correctLong = vm.locationAnswer.longitude;

correctMarker.position = mapsModule.Position.positionFromLatLng(correctLat, correctLong);

correctMarker.color = 'green';

mapView.addMarker(correctMarker);

}

showUserMarker = *function* (*args*, *answeredCorrectly*) {

*var* userMarker = new mapsModule.Marker();

userMarker.position = mapsModule.Position.positionFromLatLng(args.position.latitude, args.position.longitude);

userMarker.icon = answeredCorrectly ? 'vmarkmap' : 'xmarkmap';

userMarker.anchor = [0.5, 0.5];

mapView.addMarker(userMarker);

}

Both functions create and add markers to the map, one using the correct coordinates from the question data, and adding a normal icon, the other using the tapped coordinates and assigning the marker an icon according to whether the user answered correctly or not.

Also add these functions to the QuestionViewModel:

getLongitudeDistance = *function* (*longitude1*, *longitude2*) {

*var* rawDistance = Math.abs(longitude1 - longitude2);

return Math.min(rawDistance, 360 - rawDistance)

}

viewModel.checkMapLocationAnswer = *function* (*chosenLatitude*, *chosenLongitude*) {

*var* locationAnswer = viewModel.get("locationAnswer");

*var* distanceSquared = Math.pow((chosenLatitude - locationAnswer.latitude), 2) +

Math.pow(getLongitudeDistance(chosenLongitude, locationAnswer.longitude), 2);

*var* distance = Math.sqrt(distanceSquared);

*var* answerIsCorrect = distance < viewModel.get("errorMarginRadius");

return answerIsCorrect;

};

The second function is the one called by the question-view's code-behind 'onCoordinateTapped' function. It uses the first function to help compare the distance between the tapped location and the correct coordinates, according to the questions' data, and with the help of the question's errorMarginRadius (a tolerance factor) determines whether the user was correct or not. Computing the correct distance between two coordinates can be much more complicated (here the only "complication" taken into account is the closeness of coordinates around the antemeridian, despite their longitude value difference), but this should be good enough for the quiz's purposes.

##### Style the map

This is optional, with the style here meant to make things a bit harder for the user by removing all labels, roads and landmarks from the map. The style is taken from the file you added to the project in the beginning of this workshop. You'll need to add a 'require' in the top of the file to be able to use it:

*var* mapStyle = require("../../tools/assets/map-style.json");

Add this line to the 'onMapReady' handler:

mapView.setStyle(mapStyle);

##### Support low resolution devices

Low resolution and small devices may not have enough space to display enough of the map, which may result in the correct answer location not being displayed at all. Also, in such devices, it may be harder to tap the correct location. In this section you'll get a read of the current device's resolution and density, and change the zoom and errorMarginRadius properties of the question to take this into account.

In the question-view-model, first require the platform module, which gives access to the device's data:

*var* platformModule = require("platform");

Paste the following to the 'initQuestion' function, following the 'if (question.answers)' check:

else {

*var* screenHeight = platformModule.screen.mainScreen.heightPixels;

*var* screenWidth = platformModule.screen.mainScreen.widthPixels;

*var* screenDensity = platformModule.screen.mainScreen.scale;

*var* zoom = question.mapSettings.zoom;

*var* correctedZoom = Math.min(zoom, zoom \* (screenHeight / 1920), zoom \* (screenWidth / 1080));

*var* errorMarginRadius = question.errorMarginRadius;

*var* correctedErrorMarginRadius = Math.max(errorMarginRadius,

errorMarginRadius \* (screenHeight / 1920),

errorMarginRadius \* (screenWidth / 1080),

errorMarginRadius \* (3 / screenDensity));

viewModel.mapSettings.zoom = correctedZoom;

viewModel.errorMarginRadius = correctedErrorMarginRadius;

}

In questions that do not have an 'answers' array defined in them, i.e. Geography questions, the view model will calculate corrected zoom and errorMarginRadius, making the zoom as small as it needs to be to make sure the correct location is displayed on the map, and the errorMarginRadius larger, to help users in devices that require this.