Project 13

Using Native Controls

HTML, CSS, and JavaScript can take us a long way towards building an app that feels nearly native—it feels reasonably native, it looks nearly native, and it acts mostly native. However, if you take a closer look, you can see the small differences that tell a discerning user that things really aren't native. To help overcome this, we can blend in with our environment by using plugins that use real, native components.

What do we build?

In this project, we'll convert our path recording app built in the previous chapters into an app that uses native controls.

What does it do?

The app will do everything it did in the previous chapter; we're adding no additional features or logic. What will add are native controls: navigation bars, navigation items, bar buttons, and message boxes. We'll only focus on iOS 7 as there are only a few plugins that focus on native controls for Android (though there are a few) and most plugins have not yet been updated for iOS 7 or Cordova 3.x. This means that this app will not work on Android or iOS 6.

Why is it great?

Using native controls can provide a better sense of the app fitting in. Sometimes, it isn't possible to exactly duplicate the look and feel of a platform, and the ability to add native controls helps bridge this gap. However, they aren't a panacea, and we'll point out some of the difficulties often inherent in mixing native controls with web content.

How are we going to do it?

We'll go over these steps in order to convert the app to using native controls:

- Installing and using the plugin
- Using the navigation bar
- Using navigation items and bar buttons
- Using the message box

Installing and using the plugin

In this step, we'll cover how to install the plugin using the Cordova CLI as well as go over the overall use of the plugin.

Getting on with it

Installing the plugin is just like installing any other plugin we've used so far. Open your terminal, navigate to the project directory, and type the following:

```
cordova plugin add com.photokandy.nativecontrols
```

Once the plugin is downloaded from the Cordova plugin repository, it will be installed in your project.

In order to use the plugin, you can access it via `window.nativeControls`. This object provides access to various native controls and other utility methods that will prove useful when working with native controls.

This plugin provides access to the following native controls (and the plugin may have added more by the time you read this):

- Navigation bar (`window.nativeControls.NavigationBar`)
- Navigation item (`window.nativeControls.NavigationItem`)
- Tool bar (`window.nativeControls.ToolBar`)
- Bar button (`window.nativeControls.BarTextButton`/`BarImageButton`): These buttons are added to the preceding three native controls, and are interactive icons or text
- Message box (`window.nativeControls.MessageBox`)
- Action sheet (`window.nativeControls.ActionSheet`)

Important utility functions include the following:

- Creating rectangles (`window.nativeControls.Rect`)
- Creating colors (`window.nativeControls.Color`)

Always destroy native controls

Native controls will remain in memory once created, until they are explicitly destroyed. Unlike typical web development where JavaScript objects are garbage-collected, the native controls are not. This means you must explicitly release the memory when you no longer need them. All controls have a `destroy()` method for this very purpose. Once destroyed, set the value to null so you don't accidentally attempt to use a destroyed control.

We'll go over most of the native controls in the upcoming tasks, but let's cover using the utility functions first, since they apply to many of the controls.

Creating rectangles is important when changing the position and size of a native control. In order to create one, call `window.nativeControls.Rect`:

```
var r = window.nativeControls.Rect ( xPosition, yPosition, width,
  height );
```

If you later need to modify this rectangle, you can do so by modifying the various properties of the returned rectangle:

```
r.origin.x = 10; // left = 10px
r.origin.y = 20; // top = 20px
r.size.w = 100;  // width = 100px
r.size.h = 44;   // height = 44px
```

When changing the frames of native controls, don't make changes to the internal properties directly. This will have no effect. Instead, copy the values to another variable, make changes, and reassign it to the frame of the native control, as shown in the following code:

```
// this won't work:
someNativeControl.frame.x = 100;

// this will:
var f = window.nativeControls.Rect(someNativeControl.frame);
f.origin.x = 100;
someNativeControl.frame = f;
```

Some properties take colors, but you can't pass HTML colors or CSS RGBA colors. Instead, you need to create a color using `window.nativeControls.Color`, as shown in the following code:

```
var c = window.nativeControls.Color ( 64, 128, 192, 1 ); // red,
  green, blue, alpha
var d = window.nativeControls.Color ( "red" );
```

Once you've got a color, you can access the specific color properties directly before assigning it to a native control:

```
c.r = 128; // change red portion to 128
c.g = 192; // change green to 128
c.b = 255; // change blue to 255
c.a = 0.5; // change alpha to 50%
```

As with rectangles, don't change the colors of native controls by accessing their properties directly, or the color won't be updated. Instead, copy the color and reassign the color to the native control to update the native control, as shown in the following code:

```
var newColor = window.nativeControls.Color(someNativeControl.
someColor);
newColor.a = 0.25;
someNativeControl.someColor = newColor;
```

What did we do?

In this section, we installed a plugin that adds native controls to our app. We investigated some of the utility functions that deal with creating rectangles and colors that we'll be using with many of the native controls.

Using the navigation bar

You should already be quite familiar with the concept of a navigation bar—we've been using one at the top of our HTML views for all our projects. Now, we're going to take them out and replace them with native navigation bars.

There's no need to include a screenshot of the final result because the end result isn't much different. However, while they may not look all that different initially, once you start to interact with the control, you can immediately tell the difference. In the case of the navigation bar, iOS 7 will blur the content behind them to create a translucent effect. We can't duplicate this very well in CSS, but with native controls, the behavior is free. Likewise, the animation of the navigation bar is difficult to copy pixel-for-pixel using a CSS animation, but the native control provides it for free.

Getting on with it

Unlike the navigation bars we've used so far, iOS has a slightly different idea on how they should be built. On iOS, the navigation bar is a container that holds navigation items. These items include the title of the view and any interactive components, but without the items, the navigation bar is just a blank control on the screen.

Because of this difference, we need to manage the navigation bar and the items differently than we've been doing so far. The navigation bar will now be global to the application, and the views will push navigation items onto the navigation bar and pop items off it.

To further complicate things, we need two navigation bars for the iPad version. This means when on an iPad, the app needs to calculate their position when the orientation changes.

Let's look at the changes required in `www/js/app/main.js`. The first thing we do is declare an array to hold our navigation bars (we use an array because we need to keep track of multiple navigation bars for the iPad), as follows:

```
APP.navigationBars = [];
```

Now that we have an array, we need to create some navigation bars to go inside. This is done in `APP.start`, inside the switch statement. For iPad, we do the following:

```
var leftNavBar = window.nativeControls.NavigationBar();
var rightNavBar = window.nativeControls.NavigationBar();
APP.navigationBars = [ leftNavBar, rightNavBar ];
```

For the iPhone, we do the following:

```
var mainNavBar = window.nativeControls.NavigationBar();
APP.navigationBars = [ mainNavBar ];
```

Once we create the navigation bars, we need to set some of their properties (such as tint color and text color). We use `forEach` to accomplish this, since we need to execute the same code on all our navigation bars:

```
APP.navigationBars.forEach ( function (navBar) {
  navBar.barTintColor = window.nativeControls.Color (96, 224,
    32, 1);
  navBar.tintColor = window.nativeControls.Color ( "white" );
  navBar.textColor = window.nativeControls.Color ( "white" );
  navBar.translucent = true; // allow iOS 7 blur effect
});

…
```

Next, we will define a new method that will update the position of the navigation bars on the screen. This is called whenever the orientation changes, but it is also called when the app starts:

```
APP.updateNavigationBars = function () {
  if ( APP.navigationBars.length == 1) {
    APP.navigationBars[0].frame = window.nativeControls.Rect ( 0, 20,
document.body.clientWidth, 44 );
  } else {
    APP.navigationBars[0].frame = window.nativeControls.Rect
      ( 0, 20, 320, 44 );
    APP.navigationBars[1].frame = window.nativeControls.Rect
      ( 320, 20, document.body.clientWidth - 320, 44 );
    }
  }
```

Note that for the iPhone, we just give it a height of 44px and a width of the entire device.

For the iPad, we let the first bar take up 320px across (the width of the left-hand view), and the second bar gets to take up the remainder of the screen.

In order to ensure that this method is called when the orientation changes, we add the following to `APP.start`:

```
_y.UI.orientationHandler.addListenerForNotification
  ( "orientationChanged", APP.updateNavigationBars);
```

Of course, we also need to call it once in order to define the positions of the navigation bars when the app begins:

```
APP.updateNavigationBars();
```

Finally we need to make the navigation bars visible. If we don't do this, they will take up memory, but they won't be on screen:

```
APP.navigationBars.forEach ( function (navBar) {
  navBar.addToView();
});
```

What did we do?

At this point, if we were to run the app, it won't be very usable. For one, there's one (or two) blank navigation bars at the top of the screen obscuring our non-native navigation bars, and there's no way we can get to them. However, what we've done so far lays the groundwork to add navigation items and bar buttons so that we can interact with our app.

What else do I need to know?

Navigation bars provide several properties and methods that you can use, and we will go over them as follows:

- `addToView()`: This adds the native control to the native view and will make it visible.
- `removeFromView()`: This removes the native control from the native view.
- `destroy()`: When used with the native control, the navigation bars must be destroyed or the memory dedicated will never be released.
- `push()`: This is used to push a navigation item on the navigation bar.
- `pop()`: This is used to pop a navigation item.
- `frame`: This is used to change the position and size of the navigation bar.
- `textColor`: This is used to change the text color of the navigation bar.
- `tintColor`: This changes the tint color of the navigation bar.
- `barTintColor`: This changes the background color of the navigation bar.
- `translucent`: This determines whether the navigation bar shows the content below it by blurring it. If false, the navigation bar is opaque.

Using navigation items and bar buttons

In order to fully interact with our app, we need to create navigation items and bar buttons. The navigation items contain the title of the view and the bar buttons. These provide the interactive portions of our navigation bar.

Getting on with it

We need to remove our non-native navigation bars from our HTML templates first. Let's first look at `www/html/pathListView.html` and `www/html/pathEditView.html`. There's no code to add—we just need to remove `div` tag of `ui-navigation-bar` and its children from both files.

Next, we need to alter each view so that it creates the navigation item and any necessary bar buttons. We'll skip around a bit, so be sure to pay attention to which file we're editing. We're not going to cover every code change, so you'll want to refer to the code package for the book.

In `www/js/app/views/pathListView.js`, we create the associated navigation item and bar buttons near the end of `RenderToElement`:

```
self._navigationItem = window.nativeControls.NavigationItem();
self._navigationItem.title = _y.T("APP_TITLE");

self._addPathButton = window.nativeControls.BarImageButton();
self._addPathButton.image = "/www/js/lib/yasmf-assets/plus";

self._addPathButton.addEventListener ( "tap",
  self.createNewPath );

self._navigationItem.rightButtons = [ self._addPathButton ];
```

Notice that in the preceding code, you must specify the type of bar button you wish you can use—`BarTextButton` for buttons that have text or `BarImageButton` for icons. If you get this wrong, funny things can happen to the positioning and layout of these controls later on.

Next, we define a method that will push our navigation item we just created onto the app's navigation bar. We'll call this when the view is displayed on screen:

```
self.pushNavigation = function () {
  APP.navigationBars[0].push( self._navigationItem );
}
```

To ensure this gets called when the view is displayed, we add the following to `init`:

```
self.addListenerForNotification ( "viewWasPushed",
  self.pushNavigation );
```

While were talking about creating the navigation item and associated buttons, we should alter `destroy` to clean them up:

```
self._addPathButton.destroy();
self._navigationItem.destroy();
self._navigationItem = null;
self._addPathButton = null;
```

This takes care of almost all the changes we need to make to the list view, but we do need to change the way we interact with the edit view first. In `_displayEditView`, we make the following changes (highlighted):

```
self._displayEditView = function ( theView ) {
  if (typeof self.navigationController.splitViewController !==
    "undefined") {
    var rightView = self.navigationController.
                        splitViewController.rightView;
```

```
      if (rightView.topView.class !== "StaticView" ) {
        if (rightView.topView.isRecording) {
          rightView.topView.togglePathRecording();
        }
        rightView.topView.popNavigation();
      }
      setTimeout ( function () {
        self.navigationController.splitViewController.
            rightView.rootView = theView;
      }, 100 );
    } else {
      self.navigationController.pushView ( theView );
    }
  }
```

You'll notice that we haven't yet declared what `popNavigation` does yet; essentially it pops the navigation item that belongs to our edit view off the navigation bar. We'll worry about that in a moment. But the more important section is the `setTimeout` portion.

While a navigation bar is animating, it is absolutely critical that you avoid doing anything to it, including pushing or popping other items onto it. If you ignore this, you'll end up with very strange results. As such, we wait for a few short milliseconds until we know it is safe to talk to the navigation bar again.

We also need to do this in `deleteExistingPath`, but the code is very similar, so we won't duplicate it here.

Let's take care of `www/js/app/views/pathEditView.js`. First, in `renderToElement`, we create the native controls as shown as follows:

```
self._navigationItem = window.nativeControls.NavigationItem();
self._navigationItem.title = self._path.name;
self._navigationItem.addEventListener ("pop", self.goBack );

self._renameButton = window.nativeControls.BarTextButton();
self._recordButton = window.nativeControls.BarImageButton();
self._locateButton = window.nativeControls.BarImageButton();

self._renameButton.addEventListener ("tap", self.renamePath);
self._recordButton.addEventListener ("tap",
  self.togglePathRecording);
self._locateButton.addEventListener ("tap",
  self.centerMapAroundLocation );
```

```
self._renameButton.title = _y.T("app.pev.RENAME");
self._recordButton.image =
  "www/js/lib/yasmf-assets/circle-outlined";
self._locateButton.image =
  "www/js/lib/yasmf-assets/gps-locate";

self._navigationItem.rightButtons = [ self._locateButton,
  self._recordButton ];
self._navigationItem.leftButtons = [ self._renameButton ];
```

Next, we need to define `pushNavigation`, which is pretty similar to the last view:

```
self.pushNavigation = function () {
  APP.navigationBars[APP.navigationBars.length-1].
    push( self._navigationItem );
  // create the google map in a setTimeout using the tail end
  // of renderToElement
}
```

The first line always pushes the navigation item onto the rightmost navigation bar. On the iPhone, there's only one item, so this is pushed onto the main navigation bar. On the iPad, this targets the rightmost navigation bar.

After we push the navigation item, we move the code from `renderToElement` that creates the Google Map to this function.

Next we create `popNavigation`:

```
self.popNavigation = function () {
  APP.navigationBars[APP.navigationBars.length-1].pop();
}
```

If you remember, this is the method we were calling from the list view in order to pop the navigation off the navigation stack.

Of course, we need to update `init` to listen for the view's appearance, and we also need to destroy all our controls in `destroy`. Given that this code is very similar to what you've already seen, we won't duplicate that code here.

Finally, we need to alter `togglePathRecording` to switch the image on the recording button. The changes are highlighted in the following code:

```
self.togglePathRecording = function () {
  if (self._isRecording) {
    self._isRecording = false;
    navigator.geolocation.clearWatch( self._watchID );
    self._watchID = null;
```

```
      self._recordButton.image = "www/js/lib/yasmf-assets/circle-
outlined";
    } else {
      self._watchID = navigator.geolocation.watchPosition(
      self._updateLastKnownPosition, self._geolocationError, {
        enableHighAccuracy: true
      });
      self._recordButton.image = "www/js/lib/yasmf-assets/pause-filled";
      self._isRecording = true;
    }
  }
```

What did we do?

In this section, we modified the path list view and path edit view to create and manage the necessary navigation items and bar buttons.

What else do I need to know?

Let's go over all the properties and methods that navigation items and bar buttons support.

Navigation items have the following methods and properties:

- `destroy()`: As always, call this to remove the native control from the view and memory.
- `title`: This defines the title of the navigation item and should be the name of the view (or the document).
- `leftButtons`: This is the left-hand side group of bar buttons. Supply as many as can comfortably fit, but you will need to adjust this according to the width of the device. If the buttons take up too much space, a portion of the navigation item's title is truncated.
- `rightButtons`: This is the right-hand side group of bar buttons. Again, the only real limit is what fits visually. If the buttons take up too much space, the title is truncated.

They also fire two events:

- `pop`: This fires when the navigation item is popped from the navigation bar. This can occur when `pop()` is called on the navigation bar, or when the user physically taps the back button on the navigation bar. Since there's no way to know what happened, special care has to be taken so that the proper order of events occur regardless of whether or not the popping of the view was user-initiated or initiated programmatically.
- `push`: This fires when the navigation item is pushed onto the navigation bar.

The bar buttons have the following methods and properties:

- `destroy()`: This always cleans up after your.
- `title`: This gives the title of the button.
- `image`: This provides the image of the button. There needs to be the full relative path to the image (in our case `www/js/lib/yasmf-assets/…`). We don't need to specify the extension, as it will be inferred. We also don't need to do anything special to handle retina versus non-retina displays. If a file with `@2x` exists, it will be used for a retina display (and all YASMF's assets use this). If on a non-retina device, the same file without a `@2x` will be used; but if it doesn't exist, the `@2x` file will be resized to fit on the screen—just like we've been doing in our app so far.
- `tintColor`: This provides the tint color for the button. Leave this alone if you want to use the navigation bar's `tintColor`.

They also have the following event:

- `tap`: This is fired whenever the button is tapped

Using the message box

The YASMF framework has non-native alert boxes that look pretty good, but they aren't native. The dialogs plugin supplies native message boxes, which work well, but aren't terribly configurable (for example, one can't specify which button is the cancel button). The native controls plugin adds another method for using native message boxes. They also can act as input, which we need to have because we can't tap the navigation bar to rename the path anymore.

Déjà vu?

Does this sound a little familiar? It should! We've already covered message boxes in prior chapters. In a way, this plugin's version competes with the version supplied by the dialogs plugin, but there are some subtle differences.

Getting on with it

First, let's deal with a standard message box that only exists to present a message to the user. In `www/js/app/main.js`, in `APP.onConnectionStatusChanged`, the following needs to be done:

```
APP.onConnectionStatusChanged = function (sender, notification) {
  …
  if (friendlyMessage !== "") {
```

```
      if (APP._lastConnectionAlert === null) {
        var mb = window.nativeControls.MessageBox();
        APP._lastConnectionAlert = mb;
        mb.title = _y.T("Notice");
        mb.text = _y.T(friendlyMessage);
        mb.show();
      } else {
        APP._lastConnectionAlert.text = _y.T(friendlyMessage);
        APP._lastConnectionAlert.show();
      }
    }
  }
```

If you compare the code with the previous version of the app, it's not that much different, but the appearance is entirely native.

Let's look at the changes we need to support path renaming in the path edit view (`www/js/app/views/pathEditView.js`). It's important that we do this because the navigation bar doesn't allow us to edit the title directly. As such, we added a rename button to the navigation bar, which means we need to implement an input box as follows:

```
  self.renamePath = function () {
    var inputBox = window.nativeControls.MessageBox();
    inputBox.alertType = "input";
    inputBox.title = _y.T("app.pev.action.RENAME_PATH");
    inputBox.inputText = self._path.name;
    inputBox.addButtons ( [ _y.T("app.pev.RENAME"),
      _y.T("CANCEL") ] );
    inputBox.cancelButtonIndex = 1;
    inputBox.addEventListener ( "tap", function ( evt ) {
      var data = JSON.parse (atob(evt.data));
      if (data.buttonPressed != inputBox.cancelButtonIndex) {
        if (data.values[0].trim() !== "") {
          self._path.name = data.values[0];
          self._navigationItem.title = data.values[0];
          self.savePath();
        }
      }
      inputBox.destroy();
    });
    inputBox.show();
  }
```

The first section of code is pretty obvious—we ask for a new message box, indicate that it will be an input box, and then set its title, the initial text, and assign the buttons. We also indicate which button is the cancel button (zero-based index).

In order to receive the data about which button was pressed or the new name of the note, we attach an event handler. Due to the nature of the native controls plugin, the data returned from an input box is different than any other message box.

First, the data is encoded as a base64 string, which means it must first be decoded (hence `atob` earlier). The result is a JSON string, which we pass to `JSON.parse`. The end result is an object that looks something like the following:

```
{ buttonPressed: zero index of button pressed,
  values: [ array of input field values ]
}
```

What did we do?

We've added code to display message boxes and input boxes.

What else do I need to know?

Let's review all the properties and methods supported by message boxes:

- `destroy()`: Message Boxes don't clean up after themselves, hence this is used.
- `addButton()`: This adds a single text button to the message box.
- `addButtons()`: This adds multiple text buttons (in an array) to the message box.
- `show()`: This shows the message box.
- `hide()`: This dismisses the message box.
- `title`: This is the first line of a message box (also in bold).
- `text`: These are the remaining lines of a message box (if desired); the message box will expand to fit a reasonable amount of text.
- `cancelButtonIndex`: This is the zero-based index of the cancel button. This is returned if the message box is dismissed programmatically. This button is styled differently from the others.
- `alertType`: This is the type of message box. The normal value is `default`, which is just a regular message box. Other values are `input`, `secureInput`, and `userNameAndPassword`. The latter value provides two input fields, while the others only provide one.

- `inputText`: The text to default into the first input field. Use this only if `alertType` is not `default`, or the app will crash.
- `passwordText`: This is the text set to default into the second input field. Use this only if `alertType` is `userNameAndPassword`, or the app will crash.

Multiple events can be fired, and some of them are as follows:

- `tap`: This is fired when the user taps a button or the message box is dismissed. The data is passed in the event's `data` attribute, and it consists of the button index tapped and any input field values. The data is a base64-encoded JSON string that must be decoded first (`atob`) and then parsed (`JSON.parse`) before it can be used. The `buttonPressed` property indicates which button was tapped, and the `values` array indicates the values of the visible text fields (up to two).
- `inputChanged`: This is fired prior to the `tap` event and passes the value in the first input field as the event's `data` attribute.
- `passwordChanged`: This is fired prior to the `tap` event and passes the value in the second input field as the event's `data` attribute.

Game Over..... Wrapping it up

Congratulations! You've successfully added native controls to our web-based app. We get a lot of behavior for free now that we've added native controls, including the following:

- Blurred navigation bars of iOS 7
- Animated navigation bars (YASMF only moves them; iOS does various animations with the internal elements)
- An automatic back button on the navigation bar
- Native affordability on interactive controls, for example, bar buttons fade in and out when tapped as they will in any native app
- Navigation bars properly resize the titles and groups of bar buttons automatically
- Message boxes have smooth, natural animation
- Message boxes can also provide native input mechanisms

That said, there's also a few caveats:

- All native controls must be explicitly destroyed, or they will continue to consume memory, even when not being used. Controls used for the duration of the app (such as navigation bars) don't need to be destroyed; they will be destroyed when the app is terminated by the OS.

- Native controls live above the content. This means that the web content must still be positioned as if a non-native web control were still occupying the same space. This also means that native controls cannot be visually underneath any content—the native controls will always obscure non-native content.
- Native controls do not reposition themselves automatically. They do not use percentages like CSS can; they rely only on logical pixel values.
- Timing is important. It is a bad idea to pop and then push something on a navigation bar without waiting for the previous item to pop off first. Doing so corrupts the native control's hierarchy and strange things start happening.
- Animations can be hard to match; even though we're using native navigation bars, we aren't using native navigation controllers. Because of this, our view animates in and out using the same CSS animations it always has. This feels a little at odds with the animations provided with the navigation bar. With some effort, this can be somewhat mitigated, but it will be difficult to match the various animations exactly.
- Native controls are inflexible not because they are native, but because of the bridge between Cordova and the native world. This bridge only supports a very limited subset of the possible controls, properties, and methods, and as such, there's no ability to build an entirely custom native control without writing native code.

Even with these caveats, they can often provide native touches that users expect in their apps. When HTML, CSS, and JavaScript aren't cutting it, it's worth considering.

Can you take the HEAT? The Hotshot Challenge

There are many other native controls in the plugin; you can read the documentation for the plugin at `https://github.com/photokandyStudios/PKNativeControls` and implement the other native controls.