Mobile Foundry

Architecture Details

Draft 0.2
Feb 1, 2018

Table of Contents

Introduction

In the world of IoT, mobile applications for phones and tablets have become the de-facto user interface for setup and control of IoT devices. AylaNetworks has pioneered IoT software on all fronts, mobile, device and cloud, for a number of years. Out of this experience, Ayla has found that mobile IoT applications all share a common set of core functionality that is often difficult to write and maintain. To aid mobile developers getting applications to market as quickly and efficiently as possible, Ayla has developed a new version of its mobile application platform, Mobile Foundry.

All mobile applications will need to handle some or all of these tasks:

- User registration: new user account creation, editing
- Sign in / sign out
- Device discovery: WiFi and / or BLE
- Device registration: Adding devices to the user's account
- Device management: What devices are registered, what state are they in
- Device control: Allowing users to modify device properties
- Device schedules: Automating property changes based on time
- Device rules: Automating property changes based on other devices

Mobile Foundry provides all of this functionality and more, allowing developers to focus on the application branding and custom behavior that might require additional work.

Rather than starting from scratch, developers can define the unique aspects of their mobile application such as the look and feel, navigation style, menus, colors and images all within a single JSON configuration file. Mobile Foundry uses the information in that configuration file to configure and run mobile applications for iOS and Android. This allows developers to spend their time customizing a working application from the start rather than spending the majority of their time trying to get the application up and running in the first place.

This document describes the structure and process of the Mobile Foundry internals. While the information in this document is not required to build Mobile Foundry applications, a better understanding of the internals of the system can be very helpful to diagnose problems that may occur or explain behavior that may be confusing.

Structure

Mobile Foundry Configuration File

The heart of an Mobile Foundry application is the configuration file. A text file formatted as JSON, the Mobile Foundry configuration file contains all of the information required by the framework to build a unique application targeted at the developer's device or devices.

The Mobile Foundry configuration file contiains several sections, each filled out with information about a specific part of the application. This file is described in detail in the *Mobile Foundry Configuration Specification* document, available in the same location this document was found.

The configuration file is used both at build time (for release / stripped builds) and at run time (for all

builds). As Mobile Foundry contains many different controls, screens, images and other UI elements, including all of these in an application would be wasteful. The configuration file is parsed for release / stripped builds to ensure that only resources used by the application are included in the final build.

For all builds, the configuration file is used at run-time to initialize the Ayla SDK and provide Mobile Foundry with the details of how the application should be presented, what menus are displayed, etc.

All code running in the system is native to the platform it was built for, e.g. iOS or Android. The configuration file itself is not executable code, but rather a data structure read by the native framework on application startup to initialize the framework.

Application Code

Mobile Foundry is delivered as a standalone library. The developer is responsible for creating a simple skeleton application for each project (both iOS and Android). The application's main class (Activity or ApplicationDelegate) must derive from SepiaActivity, which allows the library a great deal of control over the application flow while still providing a means for the developer to override any behavior through subclassed methods.

An example Android application's MainActivity contains the entirety of the code required to support the Mobile Foundry application:

```
public class MainActivity extends SepiaActivity {
    private static final String LOG_TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Load the configuration
        SepiaConfig config;
        try {
             SepiaConfig config = loadConfiguration(R.raw.my_sepia_config);
        } catch (RuntimeException e) {
            Toast.makeText(this, e.getMessage(), Toast.LENGTH_LONG).show();
            return;
        }

        // Validate the configuration. This check may be removed once the
configuration has become
        // stable.
        String errors = config.validate(this);
        if (errors == null) {
            // No errors in the config. Let's get started!
            startApplication();
        } else {
            AylaLog.d(LOG_TAG, "Config failed: \n" + errors);
            Toast.makeText(this, errors, Toast.LENGTH_LONG).show();
        }
    }

    @Override
    public boolean handleDrawerMenuItem(String menuItemName) {
        // Drawer menu items that do not have an associated screen will call back
here when selected
```

```
        if (super.handleDrawerMenuItem(menuItemName)) {
            return true;
        }
        AylaLog.d(LOG_TAG, "Unhandled drawer menu item: " + menuItemName);
        return false;
    }
}
```

Here the MainActivity is a subclass of SepiaActivity. The onCreate method is overridden and handles choosing the correct configuration file to load. This particular example uses "my_sepia_config" from the setup instructions provided in the README file for Mobile Foundry.

The *loadConfiguration()* method returns an array of String values, one for each error encountered in the configuration. If no errors were found, the method returns null. If the JSON in the configuration is invalid, an exception will be thrown instead.

The application flow begins with the call to *startApplication(),* which is described throughout the remainder of this document.

SepiaActivity / SepiaAppDelegate

The SepiaActivity / SepiaAppDelegate classes provide the heart of the Mobile Foundry framework. All Mobile Foundry applications must have the main Activity or AppDelegate class derive from SepiaActivity (Android) or SepiaAppDelegate (iOS).

Structuring the application this way allows the application framework to drive the application flow while still allowing complete control and customization to the application developer. While Mobile Foundry strives to provide functionality useful to all applications, many applications will require additional screens, controls or logic in order to meet the specific needs of the developer. Because the developer will be working with a SepiaActivity or SepiaAppDelegate, any of the frameworks methods may be overridden and customized. This means if the Mobile Foundry framework does not quite meet the needs of the application, the framework's behavior can be changed at the application level very easily.

The base class contains a number of helpful methods that can be called to help with obtaining information from the configuration file, create, push or pop Screens from the navigation stack, navigate to Screens by name, etc.

Additionally, derived classes may override methods to receive events for menu taps or navigation drawer taps as well as override any of the default behavior of the framework.

Application Flow

This section describes the control flow of the application from application start-up.

Application Startup

When the application is started, the operating system loads the main Activity or UIApplication just like any normal Android or iOS application. From here, the developer calls a method of the Activity or UIApplication called *loadConfiguration*() to initialize the Mobile Foundry system with the developer's

Mobile Foundry configuraiton file. The method will return null on success, or an array of error messages if errors were found in the configuration.

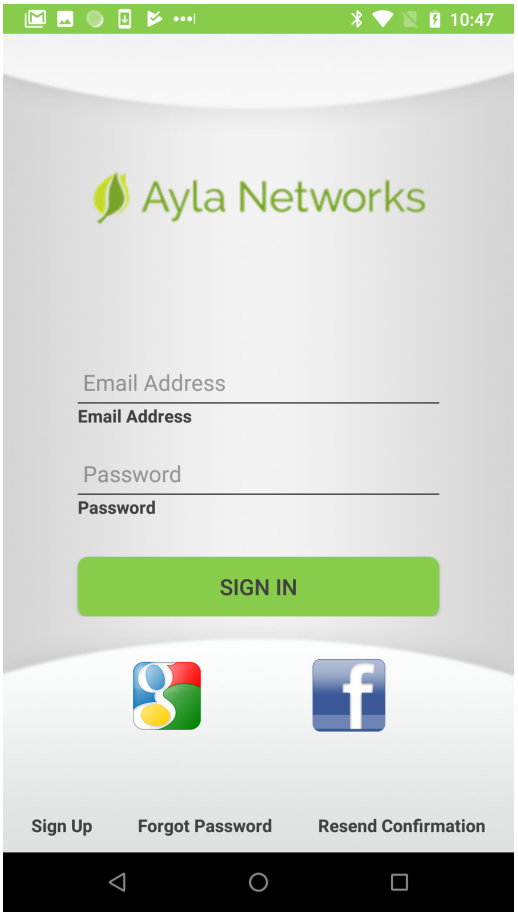
Once the configuration is loaded successfully, the developer calls *startApplication*() to hand control of the application over to the Mobile Foundry framework. The Mobile Foundry framework then takes over the control flow of the application, presenting the application interface, menus and screens based on the Mobile Foundry configuration file.

Sign In check

When *startApplication()* is called, the Mobile Foundry system initializes the Ayla Mobile SDK using values read from the configuration, and first determines whether the user is signed in, or can sign in using cached credentials (e.g. a refresh token from a previous session).

If the user does not need to provide credentials, the application authenticates with the Ayla service using the cached credentials. If the sign-in is successful, the **homeScreen** is launched. This screen is defined in the configuration in the **userExperience** section, and is required by each configuration to be present.

If the system cannot authenticate the user for any reason, the **signInScreen** is presented to the user. Like the **homeScreen**, the **signInScreen** is a required field in the configuration. Once the user has authenticated through the **signInScreen**, the **homeScreen** is presented.



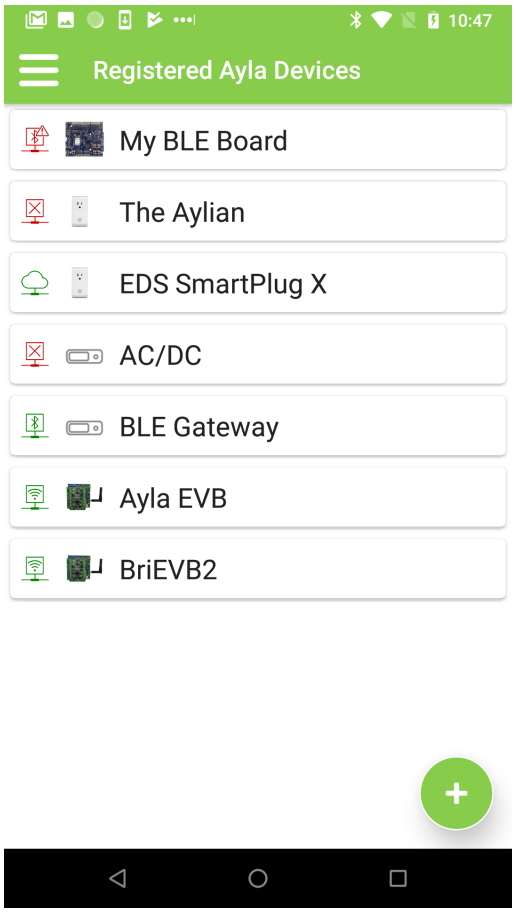


Illustration 2: Home screen (generic

The signInScreen and homeScreen are defined in the userExperience section of the configuration.

Home Screen

The first screen presented to the user after a successful sign-in is the **homeScreen**, defined in the configuration file. This screen should be considered the "main" screen of the application, and should provide the most-used functionality of the application. Mobile Foundry provides some home screens that may be used, or developers may create their own. Some home screens provided are:

AllDevicesScreen	A vertical list of all registered devices. Each entry in the list will use the **deviceControl** specified for the device as the content of the cell, if present. Tapping on a non-control area of a cell will launch the **deviceDetails** screen for the device, if defined.
DeviceSwipeContainer	Each registered device's **deviceDetails** screen is presented in a full-screen, swipeable format (swpe left / right between devices)

Additionally, developers may create their own screen classes (derived from SepiaScreen, itself a subclass of either Fragment or UIViewController). Custom screen classes are referenced identically to

built-in screen classes.

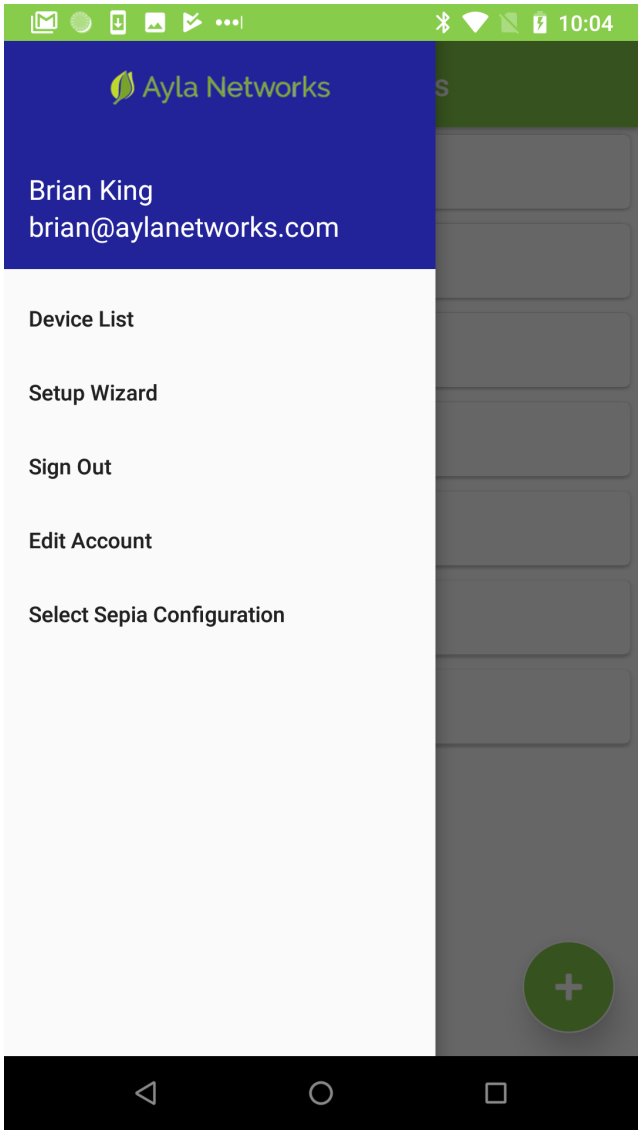
Navigation

Each application will have different requirements for navigation. Some screens or controls have pre-defined navigation behavior, such as tapping on a device in the AllDevicesScreen will launch the details screen for a particular device.

Menus

Mobile Foundry allows for the creation of menus for navigation. Menus are simple strucrues in the configuration file comprised of a name and an array of "items". Each item is a string value that may refer to a Screen name, which would result in the screen being pushed when the item is selected, or another value which is passed to the application via a call to the main activity's *handleDrawerMenuItem()* method, in the case of a drawer menu, or will call a Screen's *handleContextMenuItem()* in the case of a Screen's context menu item being selected.

When presented to the user, the menu will attempt to find the display text first from the associated Screen, if present, and if not present (in the case of an application-handled menu item) will look for an entry in the string table with the same name as the menu item. If not found, the menu item text itself will be used as a last resort.



Once defined, a menu may be attached to any Screen object by refering to it by name in the screen's "menu" field, or may be used as an application-wide drawer menu.

To enable the drawer menu in the application, simply create a Menu object in the configuration file containing a list of screens (or other strings- see below) and assign the name of this menu object to the "drawerMenu" field of the UserExperience section of the configuration:

```
"menus": [
  { "name": "drawer_menu",
    "items": [
      "device_list",
      "setup_wizard",
      "sign_out",
      "edit_account",
      "select_config"
    ]
  },
  {
    "name": "ac_details_menu",
    "items": ["device_details"]
  }
],

"homeScreen": "device_list",
```

```
    "signInScreen": "sign_in",
    "drawerMenu": "drawer_menu"
  }
```

Menu Handling

sign_out

sign_out is a special name reserved for the action of signing out of the application. Any menu item using this name will cause the application to sign the user out if selected.

When a menu item is selected by the user, either in the drawer or from a context menu, the framework first checks to see if the item is a reference to a Screen name. If so, the Screen will be created and shown to the user.

If the item does not match any Screen names, the item name is passed to the application via a call to the Activity's or UIAppDelegate's *handleDrawerMenuItem()* method in the case of a drawer menu item being selected, or a Screen's *handleContextMenuItem()* method in the case of a context menu item from a Screen being selected.

Screens and Controls

Mobile Foundry uses the concepts of Screens and Controls throughout the application. A Screen is a full-sized application window presenting information to the user. It may contain controls within it. A Control is not full-sized, and is contained within a Screen or another UI element such as a View.

Screens

On Android, Screens are subclasses of Fragments. On iOS, Screens are subclasses of UIViewController. The Screen class adds awareness of the Mobile Foundry configuration to the objects so that developers implementing Screen objects may have full access to the fields specified in the configuration file.

Controls

On Android, Controls are subclasses of ViewGroup. On iOS, Controls are subclasses of UIView. The Control class, like the Screen class, adds the awareness of the configuration parameters to the objects.

There are two basic types of Controls used in Mobile Foundry: *PropertyControls* and *DeviceControls*.

PropertyControls are tied to a particular Property of a device, and are used to display and possibly change the value of that particular property. An example of a PropertyControl is the *AylaLEDControl*, included with the Mobile Foundry distribution. The control is tied to a particular property, either Blue_LED or Green_LED in the case of the Ayla EVB. It will display in a bright color if the property value is non-zero, and a dark color if the property value is zero. It will change the value from 1->0 or vice-versa if the control is tapped. PropertyControls are attached to their property via a call to the control's *attachProperty()* method.

DeviceControls are Controls that are attached to an entire device rather than one particular property of a device. Device Controls are often composite objects and often contain *PropertyControls* within them.

Device controls represent a "quick and dirty" interface to the device, and may provide access to the more commonly-used information or controls for a device. DevicesControls are used, if present, to represent a particular Device in the *AllDevicesScreen*.

Screen and Control Configuration

The ability to customize and re-use Screens and Controls within Mobile Foundry is one of the strongest features of the framework. Each Screen or Control in the application may be used in a variety of situations and presented in a variety of formats.

Both Screens and Controls have a field called "extras" in their configuration. This field may be used to pass information to the object when it is created to customize its appearance or behavior. Multiple configurations of the same Screen or Control can exist side-by-side within a single configuration and used as independent objects.

For example, the Ayla EVB has two properties that control LEDs, one blue and one green. These properties are named Blue_LED and Green_LED respectively.

Each of these properties will use the same control, the AylaLEDControl, though one should appear blue and the other green.

To accomplish this, two Control objects are created in the configuration using the same class, AylaLEDControl:

```
{
  "name": "led_light_green",
  "class": "AylaLEDControl",
  "extras": {
    "ledColor": "green"
  }
},
{
  "name": "led_light_blue",
  "class": "AylaLEDControl",
  "extras": {
    "ledColor": "#0000aa"
  }
}
```

The "extras" field for the AylaLEDControl allows for the color to be specified via an "ledColor" value.

This essentially creates two separate controls that may be referenced by the configuration, "led_light_green" and "led_light_blue".

In order to tie the properties "Blue_LED" and "Green_LED" to these partcicular controls, we simply update the **control** field in the **managedProperties** section for our device to indicate that these controls are what should be used to represent these properties:

```
{
  "class": "AylaEVBDevice",
  "name": "Ayla EVB",
  "detailScreen": "auto_device",
  "icon": "ic_ayla_evb",
  "oemModel": "ledevb",
  "scheduleScreen": "schedule",
```

```
  "ssidRegex": "Ayla-[0-9a-zA-Z]{12}",
  "managedProperties": [
    {
      "control": "led_light_blue",
      "name": "Blue_LED",
      "notify": true,
      "schedule": true
    },
    {
      "control": "led_light_green",
      "name": "Green_LED",
      "notify": true,
      "schedule": true
    },
    {
      "control": "blue_button",
      "name": "Blue_button",
      "notify": true,
      "schedule": false
    },
    {
      "control": "generic_number_control",
      "name": "decimal_out",
      "notify": "true",
      "schedule": "false"
    },
    {
      "control": "generic_number_control",
      "name": "decimal_in",
      "notify": "true",
      "schedule": "true"
    }
  ]
}
```

In addition to providing the framework with information about the properties, specifying the actions available for a property allow the property to be used within an ActionPicker, a powerful control provided by the Mobile Foundry framework. In the below example, actions were defined for an A/C unit fan control in the configuration:

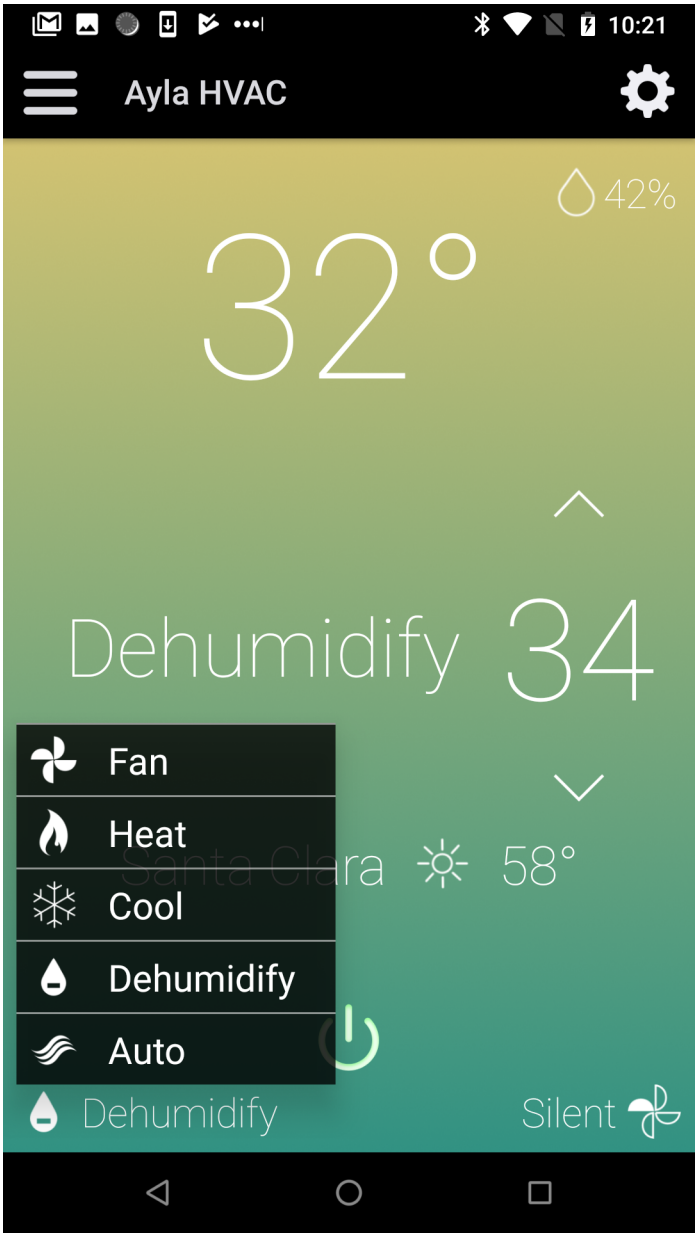
```
    {
      "name": "t_fan_speed",
      "roles": ["fanControl"],
      "notify": true,
      "schedule": true,
      "actions": [
        {
          "name": "fan_auto",
          "icon": "fan_auto",
          "value": 0
        },
        {
          "name": "fan_silent",
          "icon": "fan_silent",
          "value": 1
        },
        {
          "name": "fan_low",
```

```
        "icon": "fan_low",
        "value": 2
      },
      {
        "name": "fan_medium",
        "icon": "fan_medium",
        "value": 3
      },
      {
        "name": "fan_high",
        "icon": "fan_high",
        "value": 4
      }
    ]
  },
```

Elsewhere in the configuration, an ActionPicker was defined to use this property:

```
{
  "name": "ac_fan_picker",
  "class": "ActionPicker",
  "extras": {
    "property": "t_fan_speed",
    "textColor": "#ffffff",
    "backgroundColor": "#00000000"
  }
},
```

A Screen was created that added the "ac_fan_picker" control UI element. These linked entries in the configuration allow the UI to create the control populated with the necessary names and icons, and will automatically set the property to the desired value when an item is selected from the control's list. This



is shown in the example on the left. The example on the right shows the same control configured to use a different property, the "mode control". Same control, different configuration, different property.

Navigation

Being able to transition smoothly and easily between Screens in Mobile Foundry is a simple and easy process. Screens can be launched automatically from any menu, whether the main navigation drawer menu or a context menu on any Screen. These menus are defined in the configuration and "just work", as long as the menu names and Screen names are the same.

Additionally, any menu item that is **not** the name of a screen will be passed to the developer first to the active Screen, and then to the SepiaActivity / SepiaAppDelegate if the screen did not handle it. This allows additional functionality that might not warrant a new screen being presented to be added to the application without difficulty.

Screens themselves might contain buttons that link to other screens, launching them programatically when tapped. Examples include the default "device_details" screen, which will present a button to launch the schedule list screen when tapped. This screen allows the specific schedule list screen to be overridden in its configuration by setting the "scheduleListScreen" field to a screen name.

Developers may easily present a screen by calling one of the *pushScreen()* APIs on the SepiaAppDelegate / SepiaActivity class.

Useful Screens and Controls

Mobile Foundry ships with a set of screens and controls that may be re-used or extended. Some notable ones are listed below:

AllDevicesScreen

The AllDevicesScreen creates a list view containing items for each device registered to the account. Each cell will contain the DeviceControl for the device, if specified, or a generic control if unspecified. When tapped, the cell will launch the DeviceDetails screen for the device, if present.

DeviceSwipeContainer

Another choice for a homeScreen, the DeviceSwipeContainer presents the Details screen for each device in a swipeable format. It is highly recommended to use a context-menu navigation system over the drawerMenu system if using this as the home screen, as swiping to show the drawer menu often interferes with swiping to the next device in the list.

Fortunately moving from the drawer menu navigation to a context menu is as easy as removing the drawerMenu entry from the configuration and instead placing that menu identifier within the **menu** field of the DeviceSwipeContainer's Screen definition:

AutoDeviceScreen

The AutoDeviceScreen displays some details about the device at the top of the screen, and creates a list containing PropertyControls for each defined managed property. This screen is very useful for development, as it allows you to quickly see the defined controls for each property as well as device information automatically.

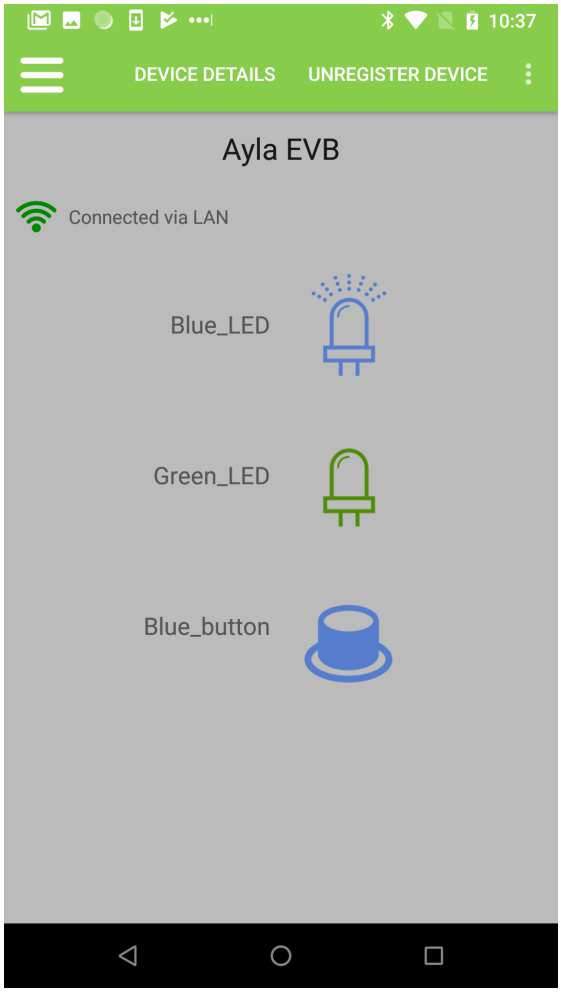


Illustration 4: AylaDeviceScreen with

AccountDetailsScreen

This screen is used both to create a new account and to edit existing account information. The **extras** field can be configured for either, as shown below:

```
      {
        "class": "AccountDetailsScreen",
        "name": "edit_account",
        "title": "edit_account",
        "extras": {"mode": "edit"}
      },
      {
        "class": "AccountDetailsScreen",
        "name": "sign_up",
        "title": "sign_up",
        "extras": {"mode": "signUp", "wantsTitleBar": false, "disableDrawerMenu":
true}
      },
```

Note that the "sign_up" version of this screen sets the mode to "signUp", which lets the class know that this is a new account being created. Also, note that the Screen extras "wantsTitleBar" and "disableDrawerMenu" are set on the version of this Screen used for signing up to ensure that the title bar is not displayed (there is none on the sign-screen in this particular configuration, so this is a better fit) and that the drawer menu is not enabled, as the user has not yet signed in and the remainder of the

applciation should not be accessible.

SignInScreen

The SignInScreen class provides the functionality to authenticate the user as well as sign up a new user. One of the only screens that is required to be defined, this will be the first screen presented to the user on the first run of the application.

Developer Workflow

This section walks through a typical developer workflow for Mobile Foundry. While Mobile Foundry does its best to provide enough to get an application up and running quickly, some application-specific details are required to be given to the framework. Additionally, most applications will wish to customize the look and feel of the app. Some others will add full Screens or Controls to the application. The amount of customization required by the developer will depend on the type of IoT device being controlled and the final look and feel of the application.

1. Get the SDK prerequisites

Each Ayla mobile application requries an app ID and app secret provided by Ayla Networks. Make sure you have these, as they are required to create accounts, sign in, etc.

2. Start a new config file, or modify an existing one

Mobile Foundry ships with a couple of sample configuration files. These are found in the common_sepia repository. If the application you are developing is similar to one of the configurations, it might be better to start with one of those. Alternately, starting a new text file is a fine option.

3. Create a skeleton application

Instructions on creating the skeleton applicaton can be found in the Getting Started Guide.

4. Set up the SDK section of the configuration

The SDK section lets the framework know about your Ayla credentials for the application, as well as what services it should talk to. A sample sdkConfig section looks like this:

```
"sdkConfig": {
    "appId": "sepiaapp-0dfc7900-id",
    "appSecret": "sepiaapp-0dfc7900-6s3Wn_kLZpbrV2ZomcCqK0EuIeQ",
    "allowMobileDSS": false,
    "allowOfflineUse": true,
    "serviceType": "Development",
    "serviceLocation": "USA",
    "consoleLogLevel": "Debug",
    "fileLogLevel": "Debug",
```

```
    "defaultNetworkTimeoutMs": 5000
  }
```

For most developers, only the appId and appSecret would be modified from this example.

5. Define your devices

In order for Mobile Foundry to work, it needs to know as much as it can about the devices it will be controlling. The configuration file contiains a "devices" section that allows developers to provide detailed information about each device, which in turn allows Mobile Foundry to control the device intelligently without (usually) requiring additional code written.

The devices section of the configuration is an array of objects, each describing a separate device. The information in this section is used when searching for, registering or generally interacting with physical devices on the users account.

Listed below is an example of the device definition for an Ayla evaluation board. The board has 2 LEDs, blue and green, that can be turned on an off. It also has a button that can be pressed on the physical board. These are mapped to "properties" called Blue_LED, Green_LED and Blue_Button.

While JSON does not allow comments, the listing below has comments added to help clarify what some of these fields mean and how they are used within AMAP.

The section below shows how the configuration can be used to assign specific controls to specific properties, how the device's SSID (for WiFi devices) regular expression can be defined to discover new devices, how "actions" can be assigned to properties and values (e.g. Mobile Foundry will know that an option for the user is to "Turn on Blue LED", and to do that it must set the Blue_LED property value to 1) and more.

```
    {
      "class": "AylaEVBDevice",     // This is the Java / Swift created to
                                    // represent the device

      "name": "ayla_evb_name",      // Entry in a string table for "Ayla EVB"
      "detailScreen": "auto_device", // Uses the auto_device screen to show details
      "icon": "ic_ayla_evb",        // Use this icon for this device
      "deviceInfo": {
        "oemModel": "ledevb"        // You can determine it's an Ayla EVB from info
                                    // included here
      },
      "scheduleScreen": "schedule", // Use this screen to manage schedules
      "ssidRegex": "Ayla-[0-9a-zA-Z]{12}", // WiFi AP looks like this
      "registrationType": "AP-Mode",  // Device uses this registration type
// Managed properties allow associating UI elements and behaviors with properties
      "managedProperties": [
        {
          "control": "led_light_blue", // Use this control for this property
          "name": "Blue_LED",          // Name of the property
          "notify": true,              // Supports notifications
          "schedule": true,            // Supports schedules
          "notification_template": "blue_led_on", // Template used to send
                                                  //notifications
```

```
        "actions": [
          {
            "name": "blue_led_on",
            "icon": "led_blue_on", // "Blue LED ON" means set this property to 1
            "value": 1
          },
          {
            "name": "blue_led_off",
            "icon": "led_blue_off", // "Blue LED Off" means set this prop to 0
            "value": 0
          }
        ]
      },
      {
        "control": "led_light_green",
        "name": "Green_LED",
        "notify": true,
        "schedule": true,
        "actions": [
          {
            "name": "green_led_on",
            "icon": "led_green_on",
            "operator": "==",
            "value": 1
          },
          {
            "name": "green_led_off",
            "icon": "led_green_off",
            "value": 0
          }
        ]
      },
      {
        "control": "blue_button",
        "name": "Blue_button",
        "notify": true,
        "schedule": false,
        "readOnly": true,
        "actions": [
          {
            "name": "blue_button_closed",
            "icon": "pushbutton_blue_pressed",
            "value": 1
          },
          {
            "name": "blue_button_open",
            "icon": "pushbutton_blue_unpressed",
            "value": 0
          }
        ]
      }
    ]
}
```

6. Define the UI

At a minimum, the **signInScreen** and **homeScreen** must be defined in the configuration. To start, the default screens ("sign_in", "device_list") should be sufficient.

Additinonaly, the contents of the drawer menu should contain at least "sign_out", giving the user a way to sign out of the application. Also recommended is the "setup_wizard" screen, which provides a starting point for setting up new devices.

At this point, the configuration should contain the bare minimum of what is required for an Mobile Foundry application. Although functional, the application will likely need additional customization in order to give it the look and feel desired.

7. UI Customization

Many parts of the Mobile Foundry-provided screens and controls allow for customization of certain aspects of their appearance or behavior. The application as a whole also defines a few specific values in the **userExperience** section that allow applicaiton-wide customization. The **defaultConfig** section allows for defaults to be provided that are visible to all screens and controls used throughout the application:

```
"defaultConfig": {
  "windowBackgroundColor": "window_bg",
  "textColor": "textColor",
  "toolbarColor": "ayla_green",
  "toolbarTextColor": "#ffffff",
  "accentColor": "ayla_green",
  "wantsTitleBar": true,
  "disableDrawerMenu": false
},
```

Colors may be either in HTML-style string format (#rrggbb) or can refer to a name from the "colors" section of the configuration.

8. Custom screens and controls

If the supplied Mobile Foundry Screens and Controls do not quite meet the requirements of your application, it is easy to add additional ones that meet your needs. Simply create a class that derives from SepiaScreen and use it just like you would any other UIViewController or Fragment. The base class will handle the reading / parsing of the configuration elements and contains methods that can be called to easily obtain these values.

Developer-defined Screens and Controls are no different from the Mobile Foundry -provided set of screens and controls, and may be used interchangably throughout the application.

9. Preparing for release

Because Mobile Foundry contains more screens and controls than most applications will use at once, built applications will be larger than necessary. To avoid this, stripping the application of unused components may be performed.

Mobile Foundry dynamically loads classes and resources based on the configuration file, which causes

problems for code-stripping tools in general. Mobile Foundry includes a Gradle plugin for Android applications that works with ProGuard, the code-stripping application that ships with Android Studio, to work around these issues.

In order for the system to know what files to keep and what to discard, Mobile Foundry needs to provide additional information to the plugin to indicate the configuration file to be used for the application as well as the location for the output file for resource keeps (keep.xml). The keep.xml file should point to a 'raw' directory in the application's resources folder.

The section in the application's build.gradle to specify this information looks like this:

```
ext {
    // The Sepia configuration file for release applications needs to be set here. This will
    // cause the Sepia plugin to use the correct configuration file when deciding what
    // elements to discard and which to keep.
    sepiaConfigFile = rootProject.file('sepia/sepia/src/main/res/raw/foundry.json')
    keepXmlFile = rootProject.file('app/src/main/res/raw/keep.xml')
}
```

Additionally, the build needs to be told to run ProGuard to perform the actual code and resource stripping. The Mobile Foundry plugin creates a file ingested by ProGuard, which must be included on the list of files ProGuard uses to process the application. A typical entry in the app module's build.gradle file looks like this:

```
minifyEnabled true
shrinkResources true
proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro',
'proguard-rules-sepia.pro'
```

The lines above instruct the build process to launch ProGuard to minify (strip unused code) and shrink resources (remove unused resources), and to use the standard Android and application ProGuard files as well as the generated 'proguard-rules-sepia.pro' file, which will contain instructions to keep items discovered through parsing the configuration file.

The code and resource stripping process is optional, and should be performed only with release builds to reduce the final application size.

As Mobile Foundry dynamically loads classes and resources by name, Mobile Foundry applications may not be obfuscated at this time.