Defense Information Systems Agency (DISA)
Global Command and Control System – Joint (GCCS-J)

Ozone Mobility Framework

Release 1.0 / v1.0 - Revision 1

11 March 2014

Prepared For:

DISA Global Command and Control System – Joint
GCCS-J Program Management Office (PMO)
6914 Cooper Ave
Fort Meade, MD 20755-0549

Prepared By:

42six, a division of Computer Sciences Corporation
6630 Eli Whitney Dr
Columbia, MD 21046

Revision History

#	Reviewer/ Org	Changes	Revision Date	Entered	Person Entering Change
1		Initial Document	30 Jan 2014	30 Jan 2014	Michael Schreiber
2		Formatting	11 Mar 2014	11 Mar 2014	Eric Chaney

Table of Contents

Introduction 4

System Architecture 4
Implementation Details 5
Activity Sequence Diagram for Authentication 7
Activity Sequence Diagram for Hardware Common API 8
Activity Sequence Diagram for App Launcher 8
Activity Sequence Diagram for App Builder 9
Activity Sequence Diagram for Apps Mall 9

Android Information 10

General Development 10
Emulators 10
User Interface 11
Application Design 11
Data Persistence 12
Testing 12
Decisions Made 12
Lessons Learned 13

iOS Information 14

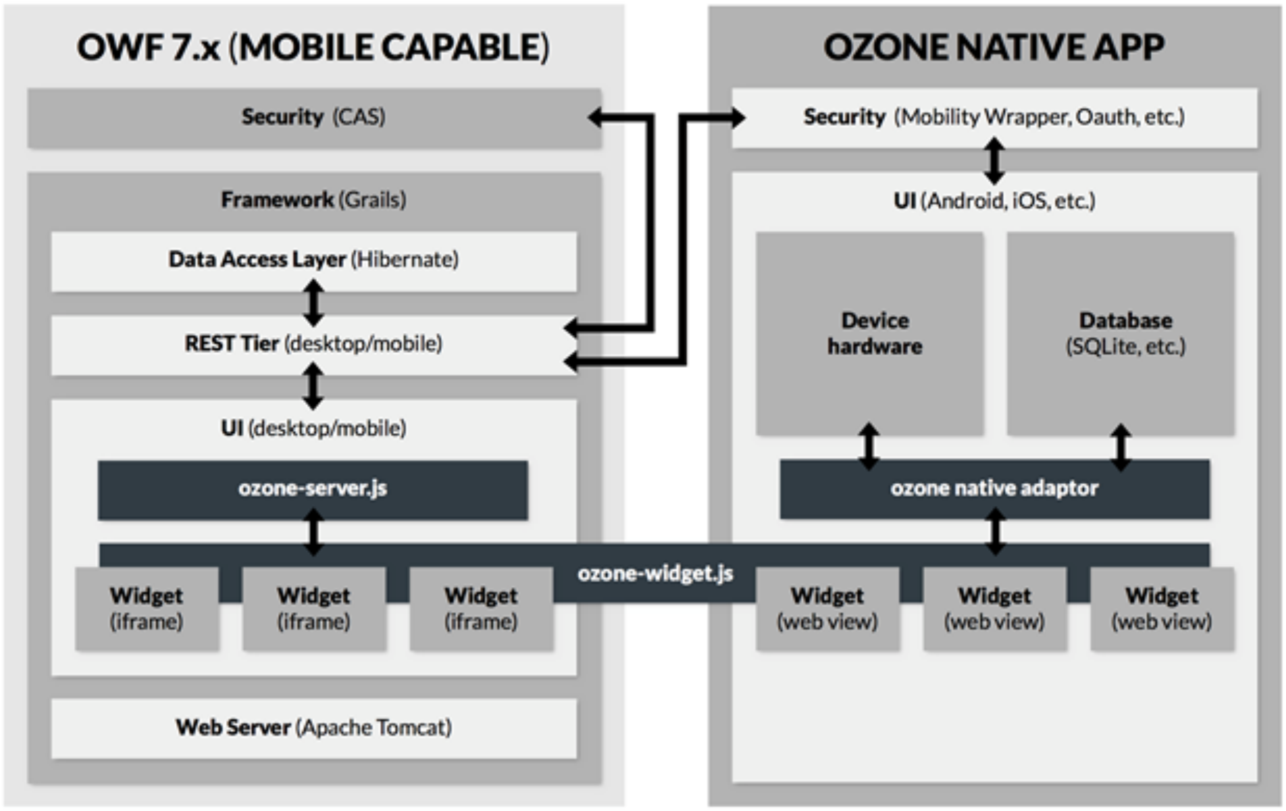
General Development 14
Emulators 14
User Interface 15
Application Design 15
Data Persistence 16
Testing 16
Decisions Made 16
Lessons Learned 16

Style Guide 18

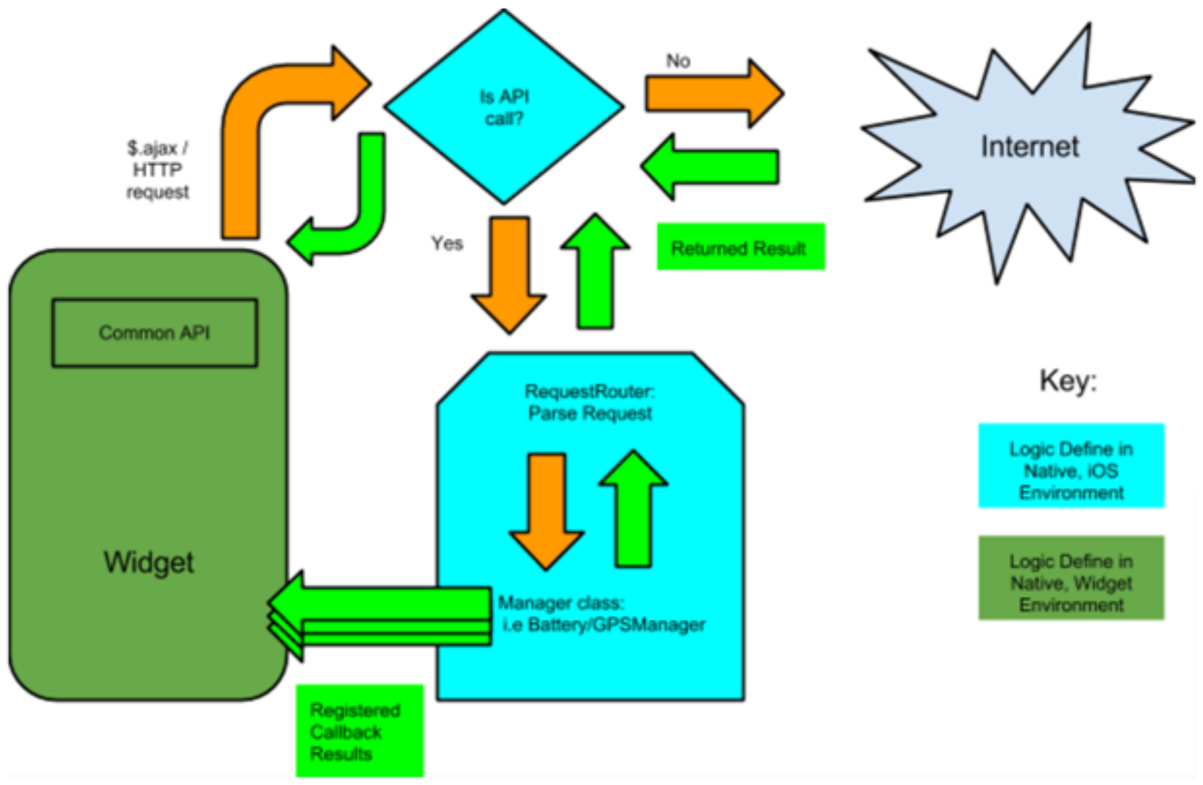
Introduction

This document contains information for the Ozone Mobility Framework on general information, development, and best practices for Android and iOS.

System Architecture



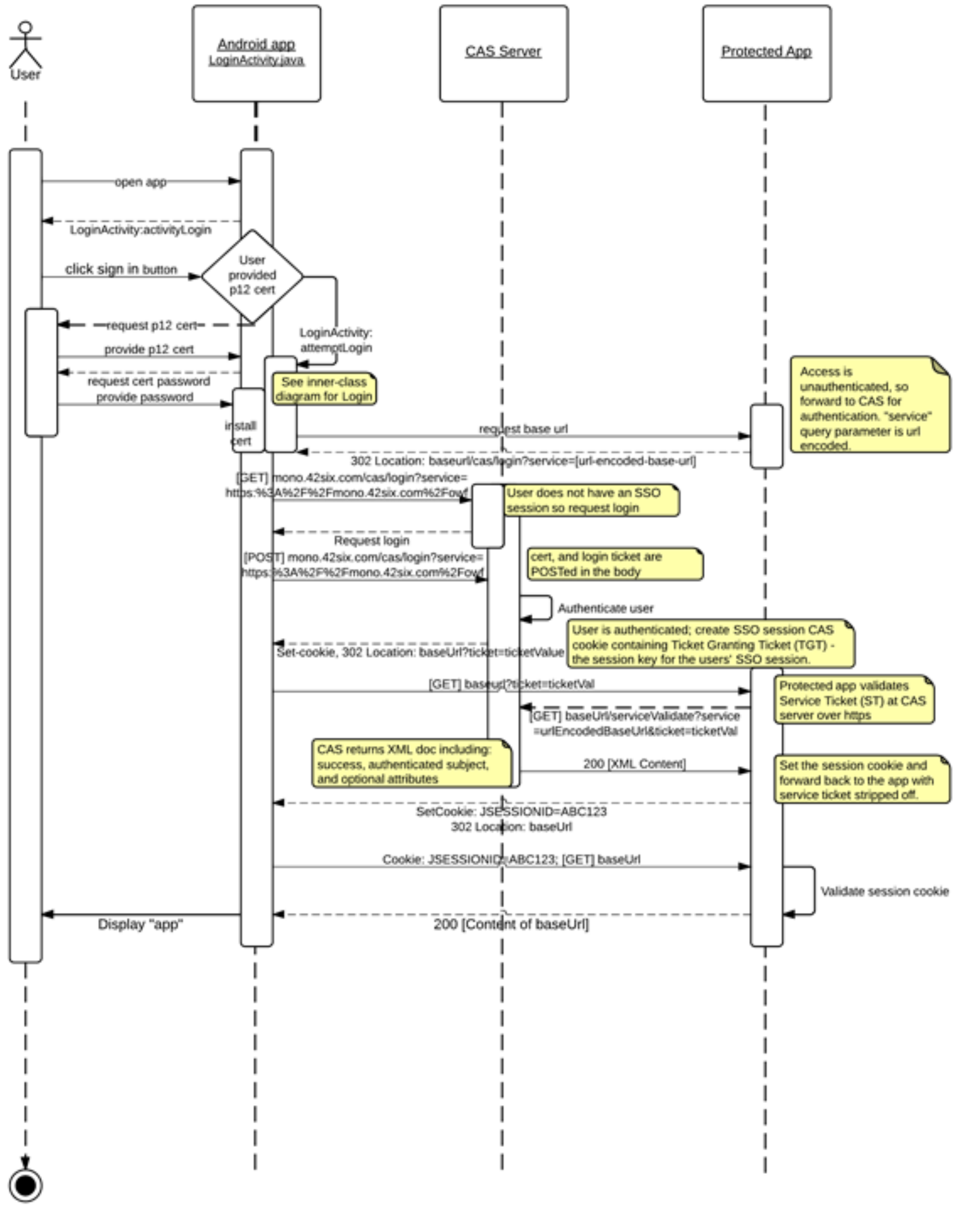
Implementation Details


$.ajax / HTTP request
Is API call?
No
Internet
Yes
Returned Result
Common API
Widget
RequestRouter: Parse Request
Manager class: i.e Battery/GPSManager
Registered Callback Results
Key:
Logic Define in Native, iOS Environment
Logic Define in Native, Widget Environment

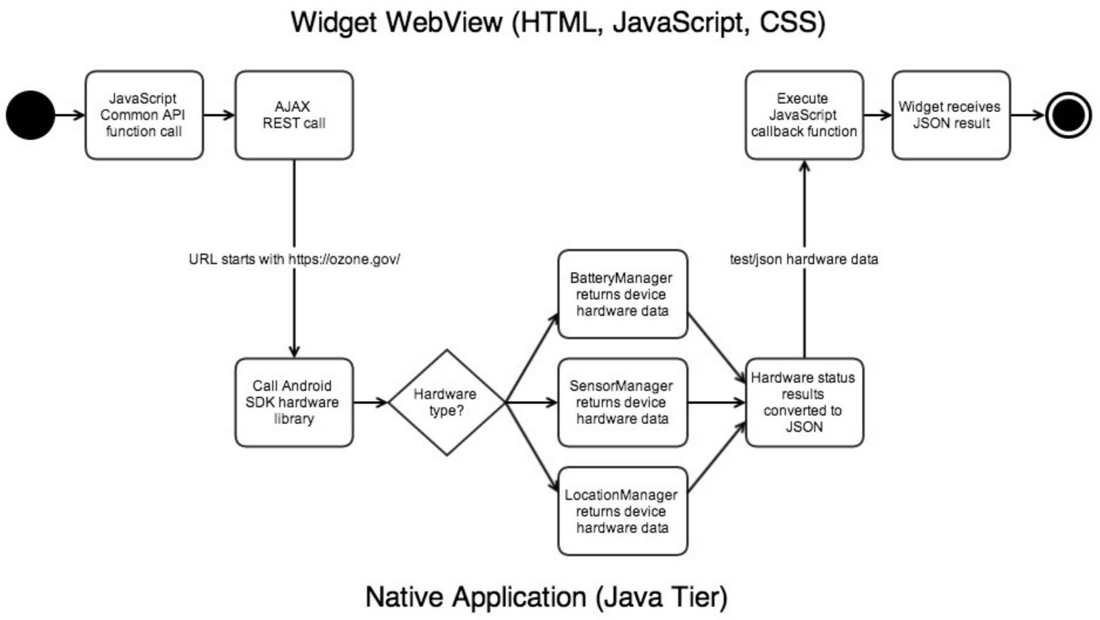
- Communicating with the native device is done through a javascript-based API.
  - The calls will originate from the widget/webview and be made via $.ajax.
  - The device will monitor outgoing HTTP calls and intercepts ones that go to the mono.gov domain.
    - Mono.gov is a sudo-domain made for the API and is subject to change. The developer doesn't need to be aware of this domain.
    - All HTTP calls currently use GET request to send data.
    - Additional information about our protocol design can be see at Communication Between Web Views and Native iOS Environments
    - The handled calls never go out to the network
- Handled Calls
  - The handled HTTP calls are parsed and extracted by a RequestRouter class.
  - The RequestRouter determines which class should process the request then sends it off to a Manager class for further processing.

- Manager Classes
  - The different components of the Common API, such as accelerometer data, GPS coordinates, battery information, etc are encapsulated by classes dedicated to managing those resources.
    - For example, we'll have a single BatteryManager class for handling all request about the device's battery status and level.
  - Additionally, the widgets/dashboards being shown on screen are managed by a WidgetManager class.
    - The WidgetManager notifies the different Managers when the user switches to a new dashboard.
    - The Manager classes use this information to cease sending data (setup via callback functions) to widgets that are no longer on screen. This helps us save battery life and reduces CPU overhead
    - Conversely, when the user navigates back to the widget/dashboard, the Managers will start sending data (setup via callback functions) back again.
- Sending Data Back to the Widget.
  - For API requests that don't subscribe to updates from the device, the data for the request is sent back as a JSON object.
    - The returned response is handled by the 'completion' handler in the $.ajax call.
    - The 'completion' handler forwards the returned JSON object to a user-defined-callback. The callback is defined by the user beforehand.
  - For API calls that register callbacks, the data is sent back directly to the user-defined-callback.
    - Callbacks must be defined with a name (i.e they can't be anonymous functions).
    - The Common API Library will run the callback inside a setTimeout call. This allows the main thread to return quickly and continue processing user events.
    - To learn more about UIWebViews and their restrictions, see iOS Lesson Learned
  - In both of the previous cases:
    - The callback function will accept 1 argument.
    - The returned object will contain key/value pairs.
    - See the Common API for details on how the object is structured.

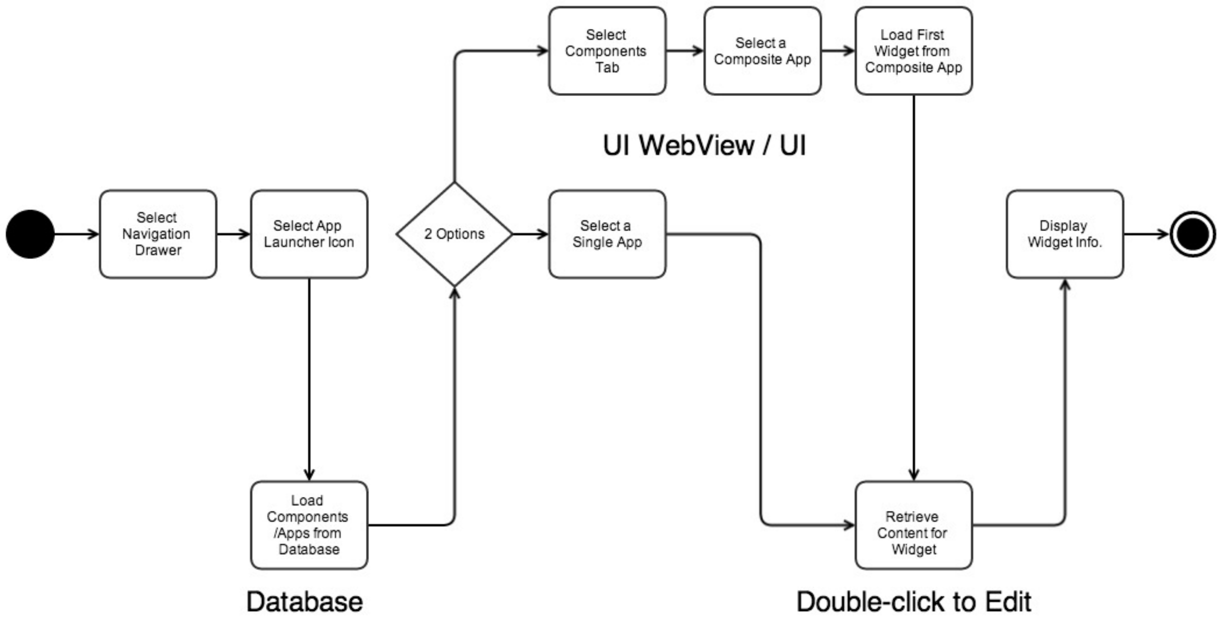
Activity Sequence Diagram for Authentication


User
Android app
LoginActivity.java
CAS Server
Protected App
open app
LoginActivity:activityLogin
click sign in button
User provided p12 cert
request p12 cert
LoginActivity: attemptLogin
provide p12 cert
request cert password
provide password
See inner-class diagram for Login
install cert
Access is unauthenticated, so forward to CAS for authentication. "service" query parameter is url encoded.
request base url
302 Location: baseurl/cas/login?service=[url-encoded-base-url]
[GET] mono.42six.com/cas/login?service=https:%3A%2F%2Fmono.42six.com%2Fowf
User does not have an SSO session so request login
Request login
[POST] mono.42six.com/cas/login?service=https:%3A%2F%2Fmono.42six.com%2Fowf
cert, and login ticket are POSTed in the body
Authenticate user
User is authenticated; create SSO session CAS cookie containing Ticket Granting Ticket (TGT) - the session key for the users' SSO session.
Set-cookie, 302 Location: baseUrl?ticket=ticketValue
[GET] baseUrl?ticket=ticketVal
Protected app validates Service Ticket (ST) at CAS server over https
[GET] baseUrl/serviceValidate?service=urlEncodedBaseUrl&ticket=ticketVal
CAS returns XML doc including: success, authenticated subject, and optional attributes
200 [XML Content]
Set the session cookie and forward back to the app with service ticket stripped off.
SetCookie: JSESSIONID=ABC123
302 Location: baseUrl
Cookie: JSESSIONID=ABC123; [GET] baseUrl
Validate session cookie
Display "app"
200 [Content of baseUrl]

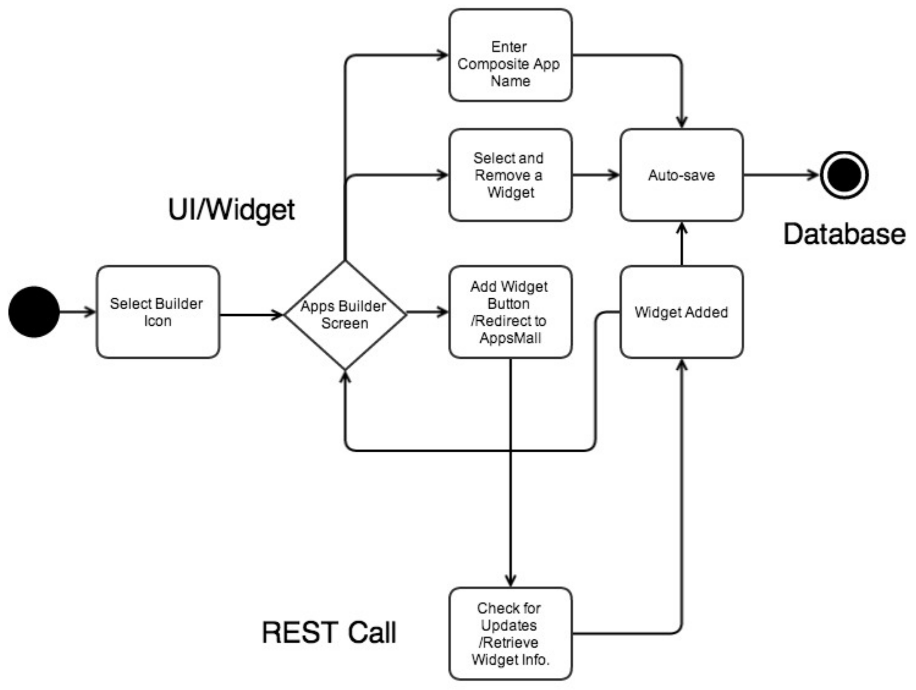
Activity Sequence Diagram for Hardware Common API


Widget WebView (HTML, JavaScript, CSS)
JavaScript Common API function call
AJAX REST call
URL starts with https://ozone.gov/
Call Android SDK hardware library
Hardware type?
BatteryManager returns device hardware data
SensorManager returns device hardware data
LocationManager returns device hardware data
Hardware status results converted to JSON
test/json hardware data
Execute JavaScript callback function
Widget receives JSON result
Native Application (Java Tier)

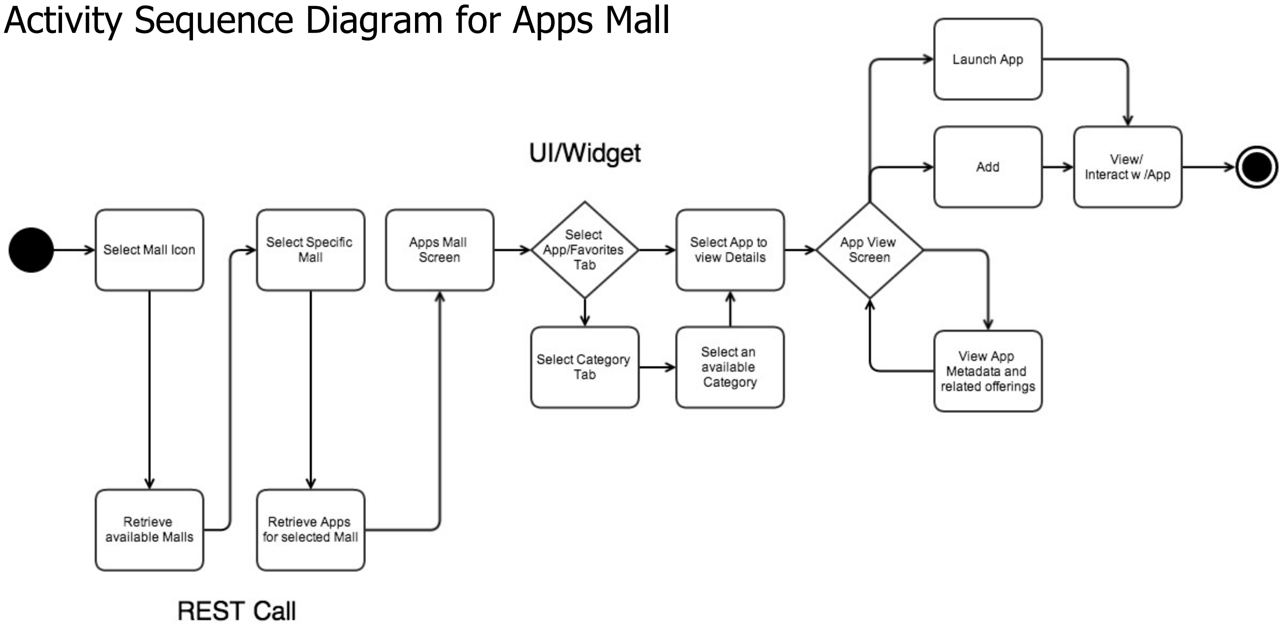
Activity Sequence Diagram for App Launcher



Activity Sequence Diagram for App Builder



Activity Sequence Diagram for Apps Mall



Android Information

General Development

The Android platform makes use of an application specific manifest file. This manifest specifies various aspects about the app that the device will need to know in order to function correctly. In it you specify things such as app info (android SDK versions, name, app theme, icon, etc), permissions (internet, camera, accelerometers, etc), and intent filters and activities. It is also possible to specify which types of android devices the app is compatible with. For example if it is meant for only higher or lower resolution screens, this is one thing that could be specified.

Development for the Android platform uses a custom Java environment modified by Google and requires the Android SDK to be installed. This comes with most of the tools you will need to create an application, one of which is the Android Debug Bridge (ADB). This tool provides various functions from connecting to the device, seeing logcat logs, deploying applications, etc. It will most likely be used by the IDE you choose to develop your application in.

Android development has two main IDEs in use today: Eclipse and Android Studio. Android Studio is based off of IntelliJ and does not require a license for use. It also gives you a powerful real time design previewer, which is helpful when designing layouts. You will be able to preview your design on multiple different screen resolutions at one time.

When it comes time to test or even deploy your application, the build process will create an APK file and your IDE will copy it over to the device and install it. It is possible to do this manually through ADB though it is easiest to just let the IDE do the work for you. Using the IDE will also make it easier to sign the application with a key upon deployment. Your application will be installed into a specific directory on the phone and given private Linux permissions to only that directory. Any files in this directory will not regularly be viewable except when the scope of your application.

Emulators

Development in Android is best done on an actual device to get a real picture of performance; however this is not always possible. You can however run an emulator when a device is not available, but they are typically slow in response and take a long time to start up. Intel has released a technology called the Intel Hardware Accelerated Execution Manager, or HAXM, which focuses on addressing this issue. With HAXM installed on a machine with an Intel processor, the emulator will experience dramatic performance increases. It is also recommended that when setting up the emulator to use the host machines GPU to help reduce the load.

The performance enhancement of HAXM does come at a small cost however. The special images used for the emulator (available through the SDK manager) are limited to specific versions of android so you will not able to test every version of Android with it. Use this link to read up on and installing HAXM: http://software.intel.com/en-us/articles/speeding-up-the-android-emulator-on-intel-architecture

It is useful to know that when deploying the application to an emulator, you do not need to restart the emulator every time. Much like a real device you can deploy the application while the system is running so you don't have to wait for the emulator to restart every time.

User Interface

Android relies solely on XML layout files for user interface design. Through these layouts you will place all of your UI elements, all of which are a subclass of "View" and most of the time end with the "View" postfix. Layouts are where you will specify things such as the elements id, layout width and height, etc. Specified view ids will be processed by the build process into a unique integer which can be referenced in the java code behind files. A simple example of this would be finding a view by its id: findViewById(R.id.foo). Using this system allows you to programmatically adjust things at runtime if need be. This same system also allows the code behind files to reference static strings stored in the res/values/ xml files via getString(R.string.foo) for example.

The android interface is designed to scale, so instead of relying on pixels, it uses what are called "density pixels." This allows the interface to provide a consistent design despite different resolutions. Due to these different resolutions, images used in android should provide multiple different sizes to account for this. When you use the images in your layouts, you do not need to specify which resolution of the image you want to use. Android will automatically search from the highest resolution down to the lowest until it finds it.

The Android user interface provides a limited selection of fonts to use. One problem with this is that users on the device can actually choose a separate font for the system. The reason this is a problem is that a different font would propagate into all applications as well. This may cause issues with rendering text in your application. This can be fixed by creating custom views in your layouts and would override the system font.

Application Design

Android applications make use of "activities." These are the base classes which are tied to the user interface to provide the code-behind functionality. Activities are started using Intents and their lifecycle can found in the android documentation here: http://developer.android.com/reference/android/app/Activity.html. Knowing the lifecycle of an Activity is important if you wish to register broadcast receivers to listen for intents.

This is also important because android will not allow any type of networking to happen on the UI thread, so that must be done in the background.

Performing networking tasks will always have to be asynchronous from the user interface thread. This is easily achieved in android with the use of AsyncTasks. They can be overridden to use either broadcast intents or handlers in order to notify the UI of the resulting data. These will run in the background using Android's thread pool, however if you wish to manage it yourself you may create your own and execute the tasks in that pool. If you need to perform long running tasks it is recommended that you move that functionality to a service as that will continue to run even if the app is stopped. Handlers are good for the UI listening to tasks whereas broadcast receivers are meant more for listening for data coming back from services. When you register a receiver make sure that you unregister at the intended place (whether in onPause, onStop, or onDestroy) in the lifecycle as it could have unexpected effects otherwise.

If you wish to start an activity with the sole purpose of gathering input from the user and returning to the current activity, you can use startActivityForResult().

Data Persistence
The android platform comes with built in support for data persistence through the use of SQLite. Using only a single connection to the database file at any given time, android provides easy to use and optimized methods for all of your SQL needs. It is highly recommended to use the built in support because of these optimizations. Those methods use prepared statements which will speed up your queries faster than using raw SQL statements. SQLite provides 4 basic data types as per the SQLite specification (http://www.sqlite.org/datatype3.html): integer, real, text, and blob. If you wish to debug SQLite and check the contents of the database, you will have to copy the file out of the app directory and use a third party app to open it. This is due to the permissions on the directory and the fact that other apps will not have access to that directory.

Testing
Android provides you with built tools for testing your applications. It comes with JUnit for testing your java code, as well as instrumentation testing for the android and user interface testing. These tests can be run on both emulators and real devices. Common tests you may want to think about are rotation of the app, UI element visibility, and random clicks/touches.

Decisions Made
1. Android Studio was used due to the powerful features from IntelliJ and ease of visual development with the layout previews.

2. The navigation for Ozone Mobile was built using the android provided navigation drawer. It provides easy to use menu functionality and using Android Studio was easy to implement. The only downside is the added error possibilities with using Fragments (see Android Lessons Learned for common issues).
3. A naming convention for files will help organize code in such a way that new developers can easily find things. One example is with layouts to prefix the name with the use of the layout such as activity_, adapter_, dialog_, fragment_, etc.
4. Images on the device will not be stored in the SQLite database. This is the recommended way as it is easier on memory and faster as it does not require conversion from bytes and extraneously stored in memory.

Lessons Learned

1. Be careful when overriding lifecycle methods in Activities, you must always have the first call inside it be a call to the superclass method, your app will crash if you don't.
2. Do not put a ListView, or anything scrollable, inside a ScrollView. Those views will handle scroll by themselves
3. The custom Java environment used in Android comes with caveats. A few Java packages have either been excluded or modified, and as such not all standard java features may exist. If you come across certain functionality not working or missing, it is possible that
4. Aside from Java classes, Android filenames CANNOT include capital letters or certain symbols even after the first character.
5. When using adapters, it is recommended to use the ViewHolder pattern. This helps eliminate calls to findViewById (much like JQuery's $("#") selector query). http://www.javacodegeeks.com/2013/09/android-viewholder-pattern-example.html
6. As mentioned earlier, networking cannot be done on the UI thread so factor out into Tasks and Services
7. AsyncTasks can be used with a ThreadpoolExecutor which can help automatically manage a threadpool and execute tasks in parallel.
8. Use android built in SQLite functionality. It will help protect against SQL injection as well as make it easier and faster to implement. Your queries will also perform much better since they will be compiled into prepared statements.
9. Android fragments have some issues with screen rotation and state saving. Every time the screen is rotated, the entire app will essentially be rebuilt. It will recreate the activities and fragments. If you don't have a default constructor in your fragments your app will crash. One way to get around this is to specify that fragments should retain their instance, and in the android manifest for that activity you would set the android:configChanges.

iOS Information

General Development

iOS applications are done almost exclusively through Apple's Xcode development platform. iOS projects can be created in the Xcode development environment, and different "targets" can be configured. In these targets, you can specify the various permissions the application will need (GPS, file sharing access, etc.), icons, and more. You can specify which iOS devices the application is designed to work with (iPhone vs. iPad).

Xcode applications, including iOS applications, primarily use Objective C. All of the other application tools and methods to debug Objective C are included in Xcode. Xcode provides methods to debug, analyze logs, and run the application in a simulated mode for development reasons, as well as deploy to development devices to test in a more realistic environment.

You can choose ARC (Automatic Reference Counting) or non-ARC development. ARC, in our opinion, is a very good approach, since it will automatically clean up unused variables once the number of references to the variable has reduced to 0. Non-ARC code would provide a greater amount of control at the expense of having to manually release every variable that is used.

In order to deploy onto actual hardware, you need an Apple Developer account, which is $99 a year. Unfortunately, there's no way around this. Deploying to a simulator does not require a developer account, but much of the hardware functionality (connectivity, battery, accelerometer, etc.) does not work properly.

To deploy your application into production, you need to create an IPA file. This can be done simply through Xcode. After creating the archive, you can select how you want to distribute it: through the app store, ad hoc (creating the IPA file), or as an Xcode archive. For our purposes, ad hoc is what we should do, as it allows us to provide our IPA to a Mobile Irons instance.

Emulators

Xcode has what it calls an iOS Simulator. The simulator provides much, but not all, of the functionality needed to test out an application. It's very fast and it's very nice to test most functionality with. There is a notable lack of hardware API support in the simulator, however. That means that trying to access the battery, accelerometer, GPS, or other various hardware sensors will not behave as expected.

There is no way to clear the internal storage of an application on iOS without completely uninstalling the application. However, the iOS simulator provides a method to completely reset it to a clean state. This is very useful when testing out functionality that may be cached.

User Interface

iOS layouts can be built using Storyboads, Nibs, or programmicaly through code. Storyboards are entirely visual, so they're fairly intuitive. Different storyboards can be created for different device form factors (iPad vs. iPhone). You can also create multiple storyboard per form factor/device. Because Apple devices are somewhat limited in their hardware configurations, it simplifies layouts somewhat as you only have to verify they work for a few different device sizes. In contrast, Nibs only display one view controller view at a time while a storyboard can display multiple.

You can layout your views in iOS using Springs/Struts or Autolayout. These frameworks allow you to automatically grow/shrink your views depending on the screen size and orientation For example when rotating, they can make the screen resizing work a lot more sensibly without a lot of work on your end.

In iOS every view on screen is a subclass of UIView. UIViews are sized on a 'points' basis rather than using direct pixels. This makes them easier to scale based on your screen size and pixel density. If you are providing custom images, it is recommend you have different images for different pixel densities (i.e. retina screens vs non-retina screens).

Application Design

An iOS application starts with a main function, and then defers all further execution to a user defined AppDelegate. From there, the application would typically spawn a View with an attached ViewController to handle program logic. Transferring between different Views is done via segues, which can be defined in the storyboard as well. ViewControllers have a set of predefined functions that can be overloaded to handle various stages of View display/execution.

Apple's guidance on iOS is to heavily favor asynchronous operations, as synchronous operations will block the main thread and prevent user input. Apple provides several mechanisms for asynchronous operation: NSOperation and Grand Central Dispatch. GCD seems to allow for a greater degree of control. For networking, using only Apple's built in asynchronous methods or a third party library such as AFNetworking, which provides only asynchronous calls is recommended.

To communicate asynchronously between different view controllers/managers, you can use the NSNotification object to publish results and subscribe to channels. This can be useful if you have something asynchronous occurring and you need to notify a view controller that it has changed or updated in some way.

Data Persistence

iOS applications have their own documents directory where they can store various information. One such use is CoreData, which is a relational store that your application can use. You can set up and generate Objective C objects, as well as the ways that they interrelate. Care must be taken when using CoreData with multithreaded applications, as changes to CoreData in one thread may not necessarily be reflected in another thread without some manual intervention.

iOS can also use the raw C sqlite library, or it can use a library like FMDB, which wraps the sqlite library in a nicer, more Objective C-like syntax. We recommend steering toward FMDB, as the sqlite library is not as easy or nice to use.

Testing

Xcode has the built in ability to write and run unit tests against your application via the XCTestCase mechanism. You can set these up pretty easily within Xcode. We recommend the use of libraries like OCMock to mock data that would be difficult to get otherwise.

Decisions Made

1. We modeled the user interface iOS application closely after the Android application, as the iOS application was developed after the Android application was well underway.
2. Objective C does not have the concept of namespaces like C++ or packages like Java, so we prepended all of our files with the three letter string “MNO” to prevent the possibility of collisions with other dependencies.

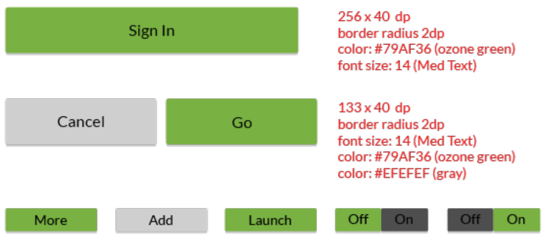
Lessons Learned

1. Despite the fact that ARC cleans up after unused variables, be aware when and where references are being captured. It’s possible to create an un-reaped reference to an object unintentionally.
2. A lot of iOS operations must happen on the main thread, which also controls the user interface. Be wary if these operations block, as it will affect the user experience.
3. The concept of the lifecycle is not as prevalent in an iOS application as it is in an Android application. Once a ViewController exists and has been created, you have a lot of power over it.
4. Many views will resize automatically very nicely, but you may need to overload resize methods to ensure that custom created items behave as expected.
5. It can be very difficult to understand the flow of how CoreData works, especially when multithreaded. Take care when using it. You may need to put CoreData operations in a NSManagedObjectContext “performBlock” function to prevent unexpected errors from arising as well.

6. Pure singletons can make testing very difficult. Consider using protocols that specify which methods a class must implement and making a mock version of your singleton for testing.
7. OCMock is capable of mocking singletons, which can make testing things like the hardware APIs much easier. That being said, it's fairly painful to make it work how you want to. Consider other options if you can.

Style Guide

Buttons



256 x 40 dp
border radius 2dp
color: #79AF36 (ozone green)
font size: 14 (Med Text)

133 x 40 dp
border radius 2dp
font size: 14 (Med Text)
color: #79AF36 (ozone green)
color: #EFEFEF (gray)

80 x 20 dp
border radius 2dp
color: #79AF36 (ozone green)
color: #EFEFEF (gray)
font size: 12 (Body Text)

Note: All buttons have box shadows on bottom edge
(black shadow opacity of 10% with vertical offset of 2dp, no blur/spread radius)

Color Palette

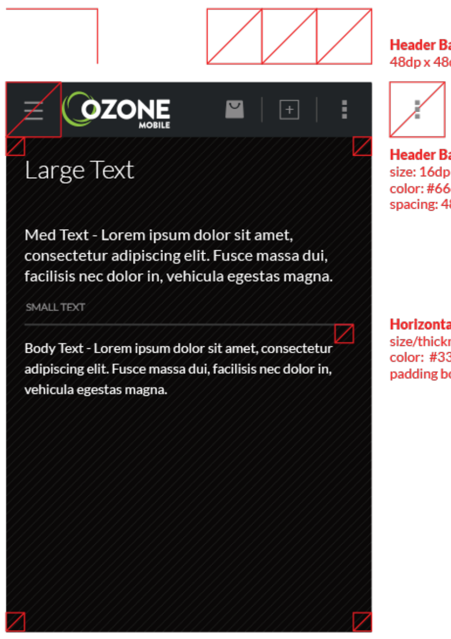
Highlight colors

⬛	⬛	🟩	⬜
#4C4F52	#5A636A	#94BE5E	#FFFFFF

Primary colors

⬛	⬛	🟩	⬜
#202427	#323D45	#79AF36	#EFEFEF

Layout & Text



Med Text - Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce massa dui, facilisis nec dolor in, vehicula egestas magna.

Body Text - Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce massa dui, facilisis nec dolor in, vehicula egestas magna.

Logo
size: 96dp x 32dp
margin top: 8dp
margin bottom: 8dp

Header Bar
height: 48dp
color: #202427

Large Text
font size: 22
color: white

Med Text
font size: 14
color: white

Small Text
font size: 9
color: #666666

Body Text
font size: 12
color: white

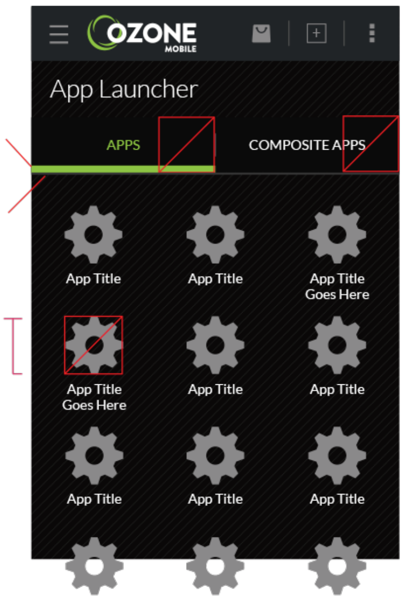
Body Content Padding
padding: 16dp

Header Bar Icon Spacing
48dp x 48dp

Header Bar Icons
size: 16dp x 16dp
color: #666666
spacing: 48dp x 48dp

Horizontal Rule
size/thickness: 1dp
color: #333333
padding bottom: 16dp

App Launcher - Grid



Tab bar (selected state)
size/thickness: 6dp
color: #79AF36 (ozone green)

Horizontal Rule
size/thickness: 1dp
color: #333333

App Icon
size: 50dp x 50dp

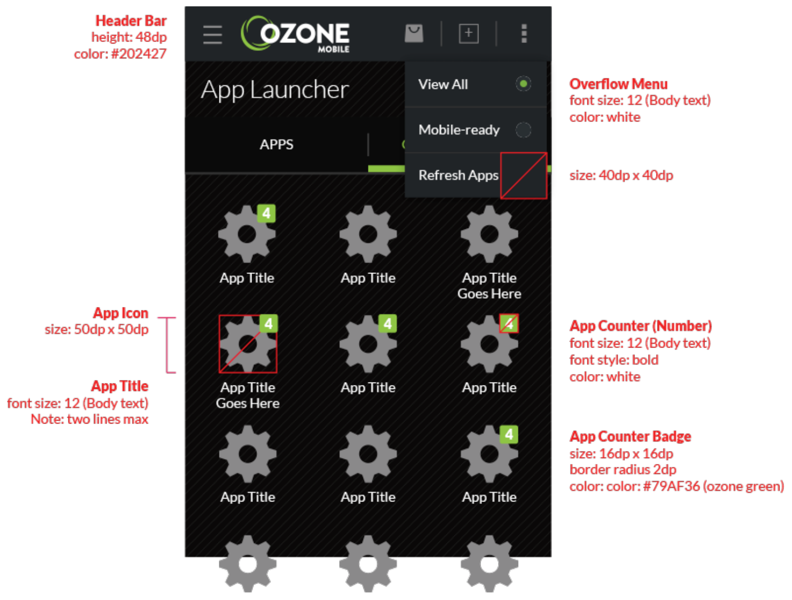
App Title
font size: 12 (Body text)
Note: two lines max

Tab
height: 48dp

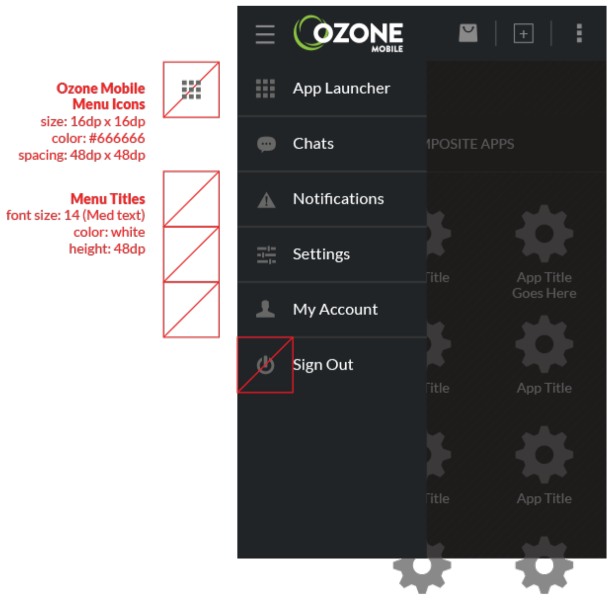
Tab Titles
font size: 12 (all caps)
color (on): #79AF36 (ozone green)
color (off): white

Grid
Portrait layout: 3 columns of App Icons
Landscape layout: allow 4 columns of App Icons

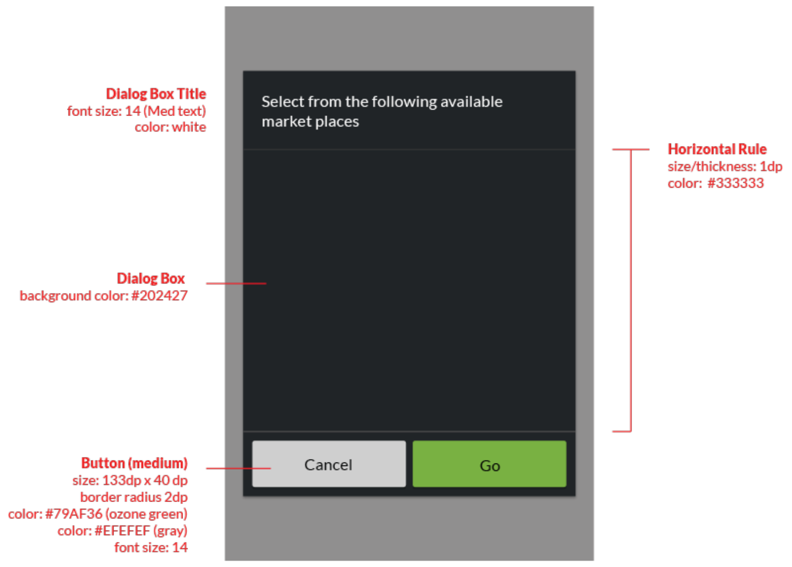
App Launcher – Action Bar



App Launcher – Drawer Menu



App View - Dialog Box



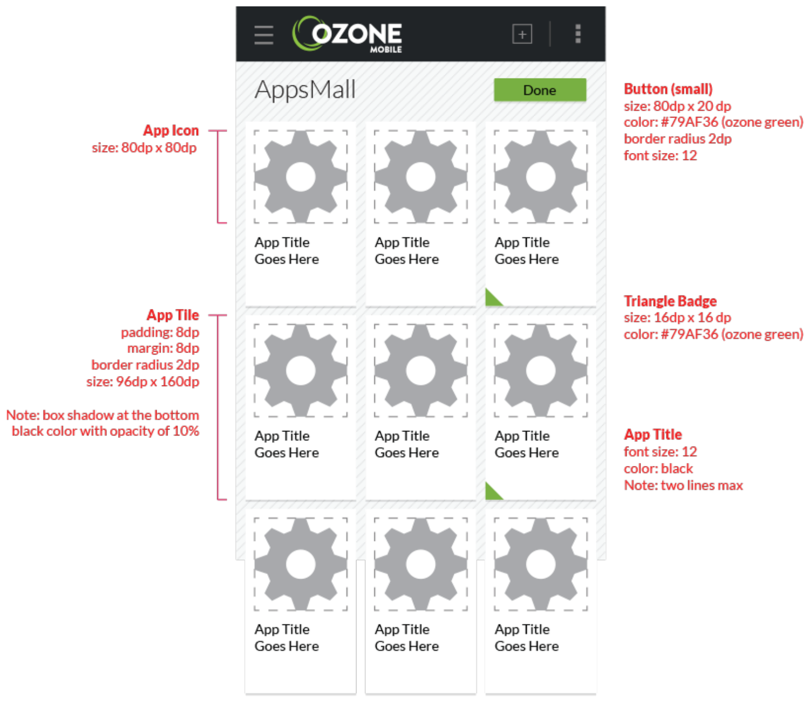
Dialog Box Title
font size: 14 (Med text)
color: white

Horizontal Rule
size/thickness: 1dp
color: #333333

Dialog Box
background color: #202427

Button (medium)
size: 133dp x 40 dp
border radius 2dp
color: #79AF36 (ozone green)
color: #EFEFEF (gray)
font size: 14

Marketplace (AppsMall)



Button (small)
size: 80dp x 20 dp
color: #79AF36 (ozone green)
border radius 2dp
font size: 12

App Icon
size: 80dp x 80dp

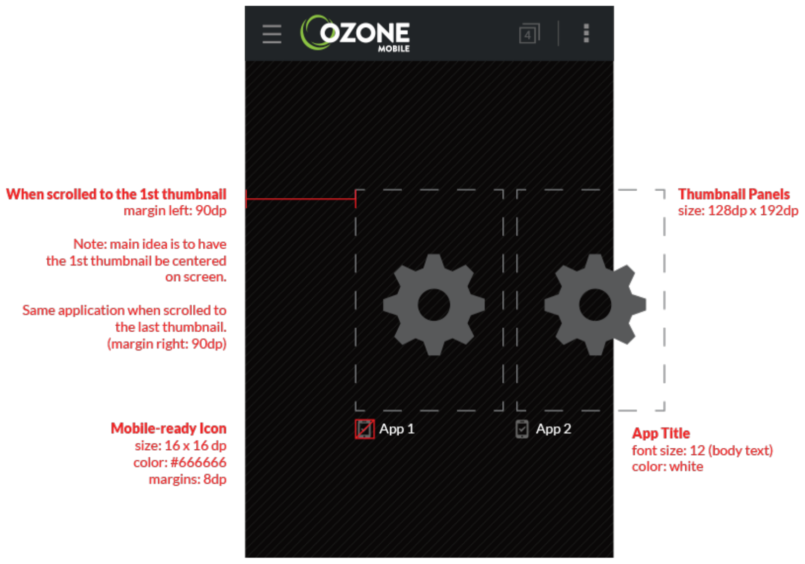
Triangle Badge
size: 16dp x 16 dp
color: #79AF36 (ozone green)

App Tile
padding: 8dp
margin: 8dp
border radius 2dp
size: 96dp x 160dp

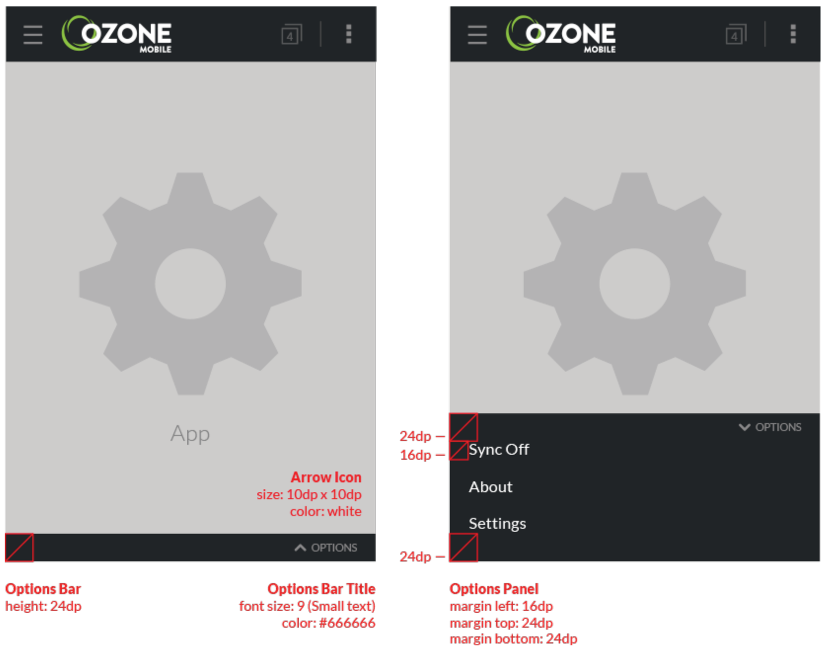
Note: box shadow at the bottom
black color with opacity of 10%

App Title
font size: 12
color: black
Note: two lines max

App View - Previewer


OZONE MOBILE
When scrolled to the 1st thumbnail
margin left: 90dp
Note: main idea is to have the 1st thumbnail be centered on screen.
Same application when scrolled to the last thumbnail. (margin right: 90dp)
Mobile-ready Icon
size: 16 x 16 dp
color: #666666
margins: 8dp
App 1
App 2
Thumbnail Panels
size: 128dp x 192dp
App Title
font size: 12 (body text)
color: white

App View – Widget Chrome


OZONE MOBILE
App
Arrow Icon
size: 10dp x 10dp
color: white
OPTIONS
Options Bar
height: 24dp
Options Bar Title
font size: 9 (Small text)
color: #666666
24dp
16dp
24dp
OPTIONS
Sync Off
About
Settings
Options Panel
margin left: 16dp
margin top: 24dp
margin bottom: 24dp