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

Ozone Mobilization Guide

Release 1.0 / v1.0 - Revision 1

25 June 2013

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	10 Mar 2014	10 Mar 2014	Michael Wilson
2		New sections added	13 Mar 2014	13 Mar 2014	Michael Wilson
3		New sections added	14 Mar 2014	14 Mar 2014	Michael Wilson
4		Design section added	14 Mar 2014	14 Mar 2014	Sam Kim
5		Updates for caching and eventing, FAQs	05 Jun 2014	05 Jun 2014	Denise Chambers
6		Updates to screenshots/graphics and Table of Content	25 Jun 2014	25 Jun 2014	Sam Kim

Table of Contents

Development Guide

Mobilization Of Your Widget

MONO provides many convenient features for mobile applications, such as:

- On device URL caching for offline widget execution
- On device arbitrary data caching
- Persistent relational database access via SQLite
- Eventing between widgets on a native device
- Access to device sensor information (battery information, geolocation information, accelerometer information)

In order to get the most out of the new mobile functionality provided by the MONO extensions to Ozone, you'll need to modify your existing widgets slightly. All attempts were made to impact existing code as little as possible.

Platform detection

A developer can easily detect if they are on a native device with a simple JavaScript function:

Mono.IsNative

  - Returns true if on a native device, false otherwise.

URL Caching & Offline Execution

MONO supports many different caching layers, many of which serve different purposes.

Native Webpage Caching

This method of caching utilizes native device browser caching. As a developer, you don't have to do anything special to access this functionality: the device's native web browser does it automatically. By default, the browser caches:

- Images
- JavaScript files
- CSS files
- HTML files

One caveat, however, is that in order for the device's web browser to cache any particular item it must first be accessed by the user. If a particular image or JavaScript file has never been accessed by the user, it will not be cached. In order to more thoroughly control what your widget caches, look into the cache manifest and MONO manifest.

Application Cache

The application cache is a caching method introduced by HTML5, and is not functionality introduced by MONO. However, since widgets are displayed via the device's native web browser, we can utilize this functionality. We highly recommend its use for caching static content, such as:

- Images
- JavaScript files
- CSS files
- HTML files

Though the device's native web browser will cache these sorts of things as well, the application cache not only makes caching more explicit, it allows caching of items that have never been accessed by the user. When preparing for offline execution, this is very important. By utilizing the application cache, you can be sure that you include all necessary static content in order for your widget to run correctly.

In order to utilize the application cache, we must create a cache manifest and reference it in our main html or view. Mono caches everything in the cache manifest, so that it will be available offline. The HTML5 cache does not cache contents of the NETWORK section.

```
CACHE MANIFEST
# 2014-03-11:v2

# Static content
CACHE:
index.html
widget-custom.js
widget-custom.css

# If the user is offline, do not load active-updates
FALLBACK:
/active-updates /offline.html

# Without this line, all not-cached files will fail to load
NETWORK:
*
```

Figure 1 - An example cache manifest

```
<html manifest=”my-example.manifest”>
    ...
</html>
```

Figure 2 – Referencing the cache manifest in your main view

Note: The URLs in the cache manifest are relative to the location of the index or page being accessed, not the location of the cache manifest itself.

Development, and widget updates

One thing to keep in mind, using this method, is that utilizing this cache can create issues when updating files. The application cache will override other methods of caching, and will need special clearing and potentially special code in order to get the most recent versions of your files. During development, this can cause issues when working on CSS and JavaScript files: the application cache will keep your old versions until it has either been cleared or updated. You can manually clear the cache using the Android Application Manager to clear data. This will empty all caches.

A web browser will only reload an application cache when the cache manifest has changed somehow. By using comments, a widget developer can force this change. With the example cache manifest above, if I were to change the comment #2014-03-11:v2 to #2014-03-11:v3, the browser will recognize the cache as “changed” and all of the files would subsequently be reloaded.

However, even though we’ve updated the cache, the browser will use the old cached files until the page has been refreshed. Fortunately, we can get information about the current state of the application cache utilizing JavaScript. The application cache has a number of events we can listen for in order to force a refresh when the browser successfully reloads the files listed in the cache manifest. In particular, we can use a JavaScript callback to listen for the application cache’s “updateready” event:

```
window.addEventListener(‘load’, function(e) {
    window.applicationCache.addEventListener(‘updateready’, function(e) {
        if(window.applicationCache.status ==
            window.applicationCache.UPDATEREADY) {
            // Perform dynamic refresh here or prompt user
        }
    }
}
```

Figure 3 – Monitoring application cache updates

This way we can ensure that updates correctly propagate to the user and still ensuring that we’ve appropriately cached content for offline usage.

Limitations of application caches

Application caches have limitations depending on the environment in which they’re being utilized. Android allows for unlimited application cache storage, though performance degrades depending on how much data is cached. iOS devices have a maximum application cache size of 10 MB.

Web server support

In order for cache manifests to work properly, the web server you're using must support displaying the "text/cache-manifest" MIME type. Enabling this will differ from server to server. However, we'll list a few common ways.

Java application servers (Tomcat, Glassfish, JBoss, etc.)

If your widget runs on a Java application server, you can enable the rendering of cache manifests by altering your application's "web.xml" file. If you add the following lines:

```
<mime-mapping>
    <extension>manifest</extension>
    <mime-type>text/cache-manifest</mime-type>
</mime-mapping>
```

Figure 4 – web.xml cache manifest modification

Apache web server (httpd)

Your Apache web server will have a configuration file called mime.types. The configuration location can vary depending on Linux distribution, installation method, and operating system. In Linux, this is often located in /etc/httpd. Once you've located the mime.types file for your httpd installation, add the following line:

```
text/cache-manifest       manifest
```

Figure 5 – mime.types cache manifest modification

Other methods

If you don't have access to the configuration of the server that your widget is hosted on, you may be able to configure a custom manifest rendering method programmatically. For example, using grails, you can ensure that any manifests within a widget will render properly with something like this:

```
class WidgetController {
    ...
    def manifest() {
        render(contentType: “text/cache-manifest”, view: “manifest”)
    }
}
```

Figure 6 – grails cache manifest programmatic example

This will vary depending greatly on which framework you're using, which type of server, and more. External research would be necessary to determine how and if such a method will work for your particular server setup.

MONO Dynamic Caching

MONO's dynamic caching is a third level of caching intended for dynamic or static content. Unlike the application cache, MONO's dynamic caching is specific to the new framework. It will not work outside of applications run outside of the Ozone Mobile application.

MONO builds a dynamic cache as widgets are accessed in Ozone Mobile. Whenever a widget accesses an external resource, static or otherwise, it is added to MONO's dynamic cache. We can be more explicit about our caching, however, by using a MONO manifest file. This file is a list of URLs to that we want to cache separated by newlines.

```
index.html
js/widget.js
css/widget.css
http://some-other-site.com/widget-data-sources.php
images/image1.html
```

Figure 7 - An example MONO manifest

In order to reference the MONO manifest, we need to add a special attribute to the HTML tag in our index or main view:

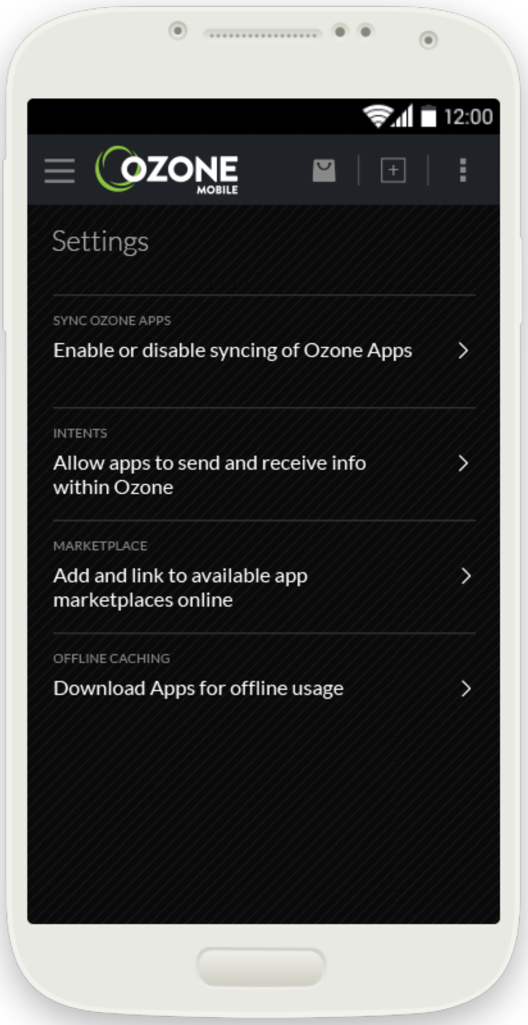
```
<html data-mono=”my-example-mono.manifest”>
    ...
</html>
```

Figure 8 – Referencing the MONO manifest in your main view

Note: The URLs in the MONO manifest are relative to the location of the index or page being accessed, not the location of the MONO manifest itself.

MONO Aggressive caching

The manifest is read when the user prepares for offline usage in the Ozone Mobile by clicking the "**Start**" button at the bottom of the settings screen:



Settings

SYNC OZONE APPS
Enable or disable syncing of Ozone Apps

INTENTS
Allow apps to send and receive info within Ozone

MARKETPLACE
Add and link to available app marketplaces online

OFFLINE CACHING
Download Apps for offline usage

Figure 9 - Ozone Mobile settings page with offline usage

Once this button is pressed, Ozone Mobile will go through every widget the user has access to and read the manifest if available. It will then go and cache each URL listed in each manifest.

Additionally, every time a URL is accessed, it is put into the dynamic cache. This includes static content such as images, CSS, JavaScript files, etc. While this is somewhat redundant with the native browser cache, it is much more explicit, and provides us with some assurance that our data is being cached at some layer.

If Ozone Mobile has cached a file or URL, it will retrieve the file from the dynamic cache.

Dynamically generated manifests
The manifest does not need to be a static file. In fact, for dynamically generated content, it is encouraged to point the manifest attribute at a dynamically generated endpoint that outputs a list of URLs to cache within the Ozone Mobile application.

As an example: We have a widget that hits an endpoint, http://remote-server/targetList that outputs a JSON formatted list of target identifiers.

```
{“targetIds”: [“1”, “2”, “99”, ...]}
```

Figure 10 – JSON output of the example REST endpoint

The widget uses this list to gather information about each target by accessing another endpoint, http://remote-server/targetInfo with the various identifiers gathered from the previous step as GET variables.

```
http://remote-server/targetInfo?id=1
http://remote-server/targetInfo?id=2
http://remote-server/targetInfo?id=99
```

Figure 11 – Endpoints generated from the list of target identifiers

Ideally, we’d be caching each of these URLs for offline usage. Utilizing a static manifest, we’d have no way of gathering the list of target IDs used to generate the targetInfo URLs, and no way of generating that arbitrary list without regularly updating themanifest by hand. Instead of doing that, we can make a grails controller that accesses the targetIds endpoint, parses the JSON, and generates the list of targetInfo URLs to cache.

On Device Data Caching

The MONO framework allows for widgets to cache application data on the native device. This differs from URL caching in that this is not the caching of web pages or endpoints, but rather any application specific data that a widget developer would consider useful. In order to do this, there are several JavaScript calls that are available for use.

JavaScript Functions

There are three JavaScript functions that widget developers can utilize for data caching:

- Mono.Caching.Initialize
    - This function will create and initialize a cache to allow for the storage and retrieval of data. It will execute a callback and pass in an identifier for created cache.
- Mono.Caching.Store

  - This function will store data into an initialized cache given a cache identifier. It will execute a callback and pass in an identifier for the newly stored data.
- Mono.Caching.Retrieve
  - This function will retrieve data from an initialized cache given a cache identifier and a data identifier. It will execute a callback and pass in the data requested from the cache.

For more information on the various function calls, including parameters and other details, please refer to the MONO JavaScript API Documentation.

Persistent Relational Database Storage

As of now, the MONO framework supports native device relational storage. Widgets that use relational storage can only access their own personal databases. They cannot access the databases of other widgets. The underlying implementation of the relational database is SQLite3.

JavaScript Functions

There are two JavaScript functions that widget developers can utilize for utilizing relational storage:

- Mono.Storage.Persistent.Exec
  - This function is intended for database queries that require no results. This is best used for create statements, update statements, insert statements, etc. It will execute a callback and pass in the success value of the statement.
- Mono.Storage.Persistent.Query
  - This function is intended for database queries that are expected to return results. This should be used for select statements in particular, as this will return all rows from a query that was issued. It will execute a callback and pass in the resulting rows from the query.

For more information on the various function calls, including parameters and other details, please refer to the MONO JavaScript API Documentation.

Eventing On A Native Device

With the MONO framework, eventing works just like it does on the desktop. No code changes are required, as MONO overloads Ozone's default eventing functions so that they support both the original eventing functionality and new native functionality.

One note about eventing: widgets can only event to one another in a composite application or desktop loaded from OWF. Widgets that are run singularly cannot event to other widgets.

Due to the vast differences in browser events, use a common eventing API that works across mobile browsers, desktop browsers, as well as web views within apps. For this, jQuery Mobile Events API can be found at: http://api.jquerymobile.com/category/events/

Device Hardware Access

MONO also allows applications to access a native device's hardware information. At the moment, we support accessing:

- Battery level and charging state
- Geolocation information
- Connectivity information
- Accelerometer changes

Battery information

MONO has several methods for accessing battery information.

- Mono.Battery.GetBatteryPercentage
  - This function passes the current battery percentage level to a provided callback.
- Mono.Battery.GetChargingState
  - This function passes the current charging state to a provided callback. Charging states can be:
    - Charged
    - Charging
    - Discharging
    - Unknown
- Mono.Battery.RegisterForUpdate
  - This function allows for regularly occurring updates as to the current battery status. It will regularly execute a provided callback with both the current battery percentage and charge state supplied.

Geolocation information

Widget developers can interrogate the device for the current device GPS coordinates, as well as registering for regular location updates.

- Mono.Location.GetCurrentLocation
  - This function detects the current device latitude/longitude and passes the information to a provided callback.
- Mono.Location.RegisterForUpdate
  - This function registers for regular geolocation updates from the native device. It will execute a callback every time the user steps travels a developer defined radius from the last update.

- Mono.Location.DisableLocationUpdates
  - Unregisters from updates that were subscribed to with the previous function.

Connectivity information

The connectivity of the device can be interrogated through the use of several JavaScript functions.

- Mono.Connectivity.IsOnline
  - This function will execute a callback with information as to whether the device is currently online.
- Mono.Connectivity.RegisterForUpdate
  - This function will execute a callback every time the device's connectivity state changes.

Accelerometer changes

At the moment, MONO supports the ability to monitor accelerometer changes at a regular rate.

- Mono.Accelerometer.RegisterForUpdate
  - This function will execute callbacks at a regular interval that will be given the current accelerometer information for the device.
- Mono.Acclerometer.DisableUpdates
  - This function will disable reception of updates from the previous function.

Device Hardware Graphics Acceleration

On Android, the graphics acceleration hardware is automatically enabled by the native container. On iOS, graphics acceleration is not automatically enabled. In iOS, some CSS elements that enabled graphics acceleration in iOS 5 don't anymore in iOS6. Certain CSS features must be utilized to ensure graphics hardware acceleration. Here are some links that describe the issues.

http://stackoverflow.com/questions/16185639/list-of-hardware-accelerated-css-properties-for-mobile-safari

http://indiegamr.com/ios6-html-hardware-acceleration-changes-and-how-to-fix-them/

http://stackoverflow.com/questions/12529286/ios6-uiwebview-css3d-transforms-are-not-hw-accelerated

Frequently Asked Questions

1. How do I know if my widget is running in a browser or on a device?

    Mono API function: Mono.IsNative()

2. What includes do I need to add to my widget to support Mono calls?

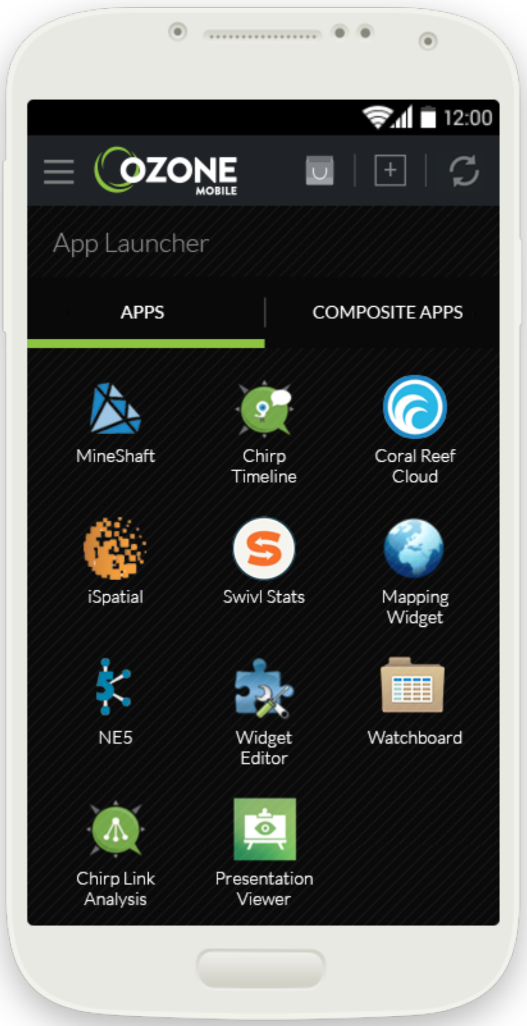
    owf-widget-min.js

Design Guide

Icons & Images

Icons can be great in that they take up a small portion of screen real estate and provide a quick, intuitive representation of an action, a status, or an app. Images can also enhance the UI of apps and give them distinct looks and styles.

When using icons and images, keep in mind that your app may be installed on a variety of devices that offer a range of pixel densities (iOS retina display). You can make your icons look great on all devices by providing rich, high-res icons and images that are resizable.



OZONE MOBILE
512 x 512

Launcher icons for the AppsMall
512 x 512 pixels (standard resolution)

App Icon for the App Launcher
We recommend icons to be of at least 320 x 320 pixels (ideal resolution compatible for Android and iOS)

Utilize resizable, rich, high-res graphics in PNG (Portable Network Graphics) format.

Avoid Platform specific icons and graphics. Many of Android/iOS/Windows platform symbols are copyrighted, and product designs can change frequently.

These Ozone Mobile's high-res assets are saved as 320 x 320 pixel PNG files and can be resized to fit on a variety of devices that offer a range of pixel densities.

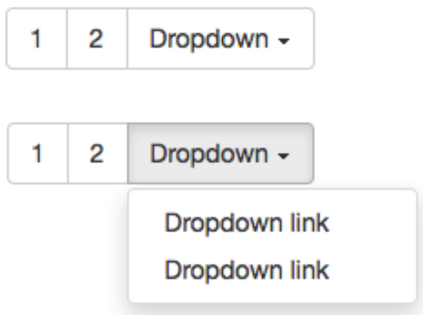
Android App Iconography - http://developer.android.com/design/style/iconography.html
iOS Icon and Image Sizes -
[https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/IconMatrix.html - //apple_ref/doc/uid/TP40006556-CH27-SW1](https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/IconMatrix.html)

Intuitive Navigation

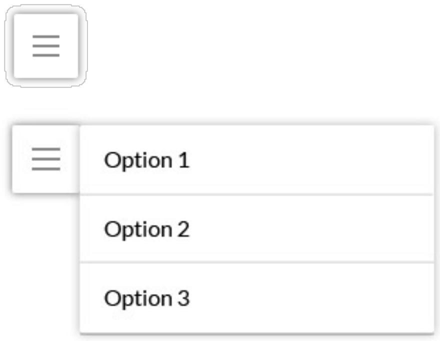
Make navigation intuitive - Design well-defined, clear task flows with minimal navigation steps, especially for major user tasks.

One intuitive and efficient entry way to an app menu on a mobile device is through the usage of a fly-out menu; also known as a "hamburger menu."

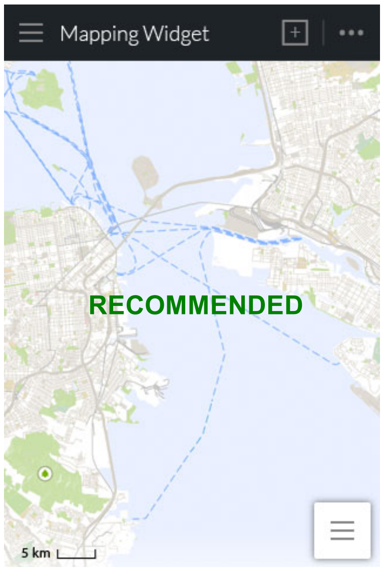
Dropdown Menu

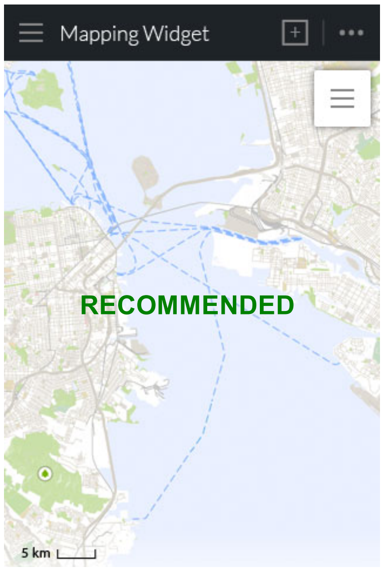


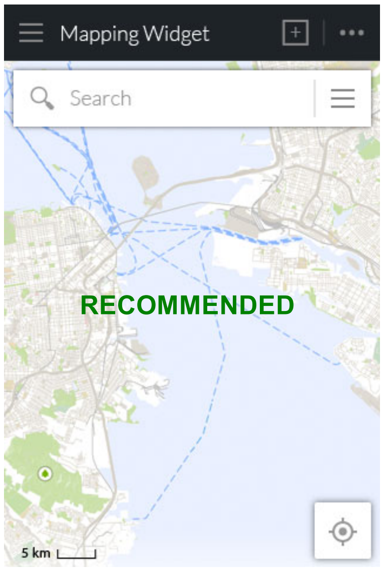
Fly-out/Hamburger Menu

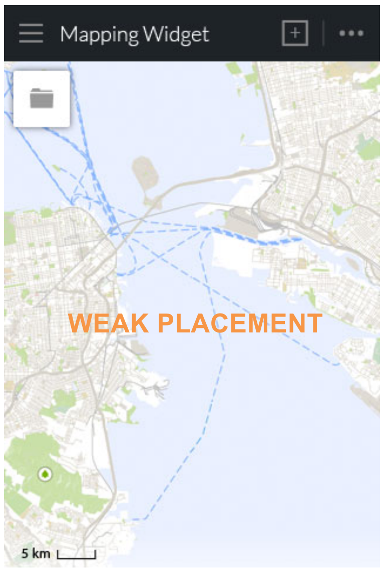


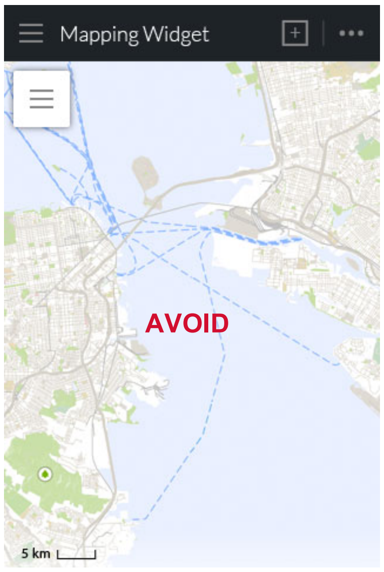
Navigation Menu Placement











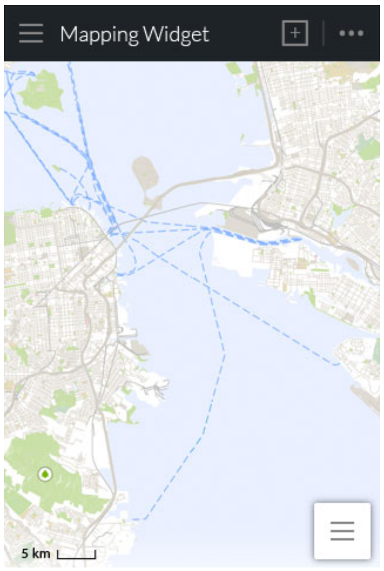
Note: Since the Ozone Mobile's navigation bar will always be docked at the top of the device screen, we recommend avoid placing a hamburger menu at the top left corner of the available screen space as this menu will compete with the Ozone Mobile's main drawer menu.

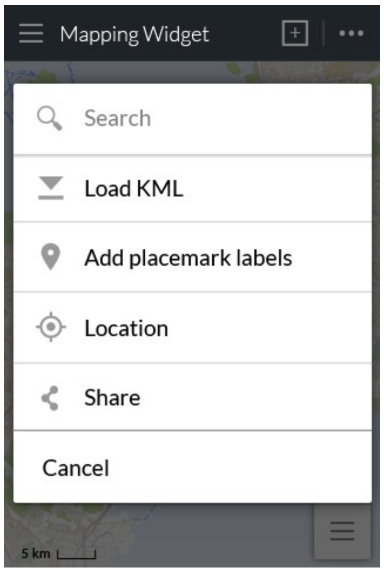
Using modals/dialog tasks

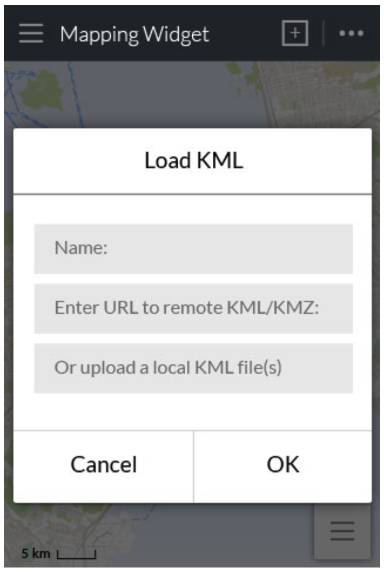
Dialog tasks - Keep the widget configuration light and don't present more than 2-3 configuration elements. Use dialog-style instead of full-screen activities to present configuration choices and retain the user's context of place, even if doing so requires use of multiple dialogs.

Keep dialog tasks simple, short, and narrowly focused. You don't want your users to experience a dialog view as a mini app within your app. If a subtask is too complex, people can lose sight of the main task they suspended when they entered the dialog context. Be especially wary of creating a dialog task that involves a hierarchy of views, because people can get lost and forget how to retrace their steps. If a dialog task must contain subtasks in separate views, be sure to give users a single, clear path through the hierarchy, and avoid circularities.

Always provide an obvious and safe way to exit out of a dialog task. Users should always be able to predict the fate of their work when they dismiss a dialog view.







The hamburger menu acts as an entry way into the Mapping Widget's menus.

Users can navigate deeper into the menu selections or click cancel to exit out and return to the previous screen.

Touch-friendly Experience

We recommend that developers design apps with the expectation that touch will be the primary interaction method of your users. The following are some key pointers to keep in mind while designing for a touch-friendly experience.

Provide enough room for users' fingers

- Larger target areas for command buttons for touch-friendly experience.
- Provide enough spacing/padding around the target areas.

Recommended touch target sizes:

- Android Metrics and Grids recommends a touch target size of 48 pixels
- Apple's iPhone Human Interface Guidelines recommends a minimum target size of 44 pixels wide 44 pixels tall.
- Windows 8 touch interaction design guidelines recommends a target of at least 7mm (about 40px) square with at least 2mm (about 10px) of padding around it.

Make links/buttons, target areas obvious

- By association - applying the same color and/or text style on the touch targets, ie. underlined or bolded blue text links.
- Style of verbiage - using short, active verbs, and common nouns.

Note the basic gestures in touch devices

Mobile Devices	**Desktop**
Tap	Left mouse click
Tap & hold (long hold)	Right mouse click
Swipe up/down	Scroll wheel on mouse/Scroll bar (vertical scrolling)
Swipe left/right	Side scroll (Mac mouse)/horizontal scroll bar (side scrolling)
Pinch	Scroll wheel on mouse (vertical scrolling)

Note: there are no double-clicking functionality on mobile devices.

Layout Considerations

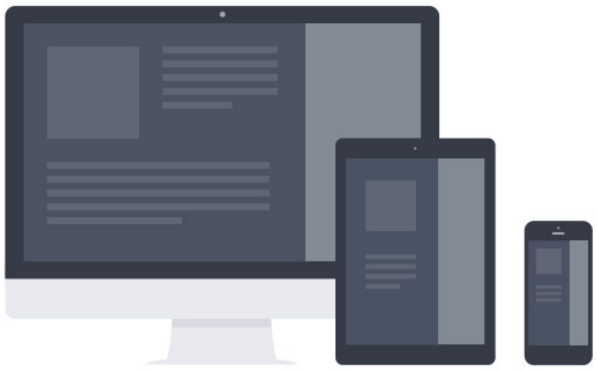
Goal is to reach optimal viewing experience - One of the most important UI issues to consider when creating an app is how to respond to different screen sizes as well as adjust to screen rotations. Just as we recommend developers to design apps with the expectation that touch will be the primary interaction method of their users, in designing a responsive app, we recommend that developers start from the smaller screen dimensions and work their way up to the desktop layout.

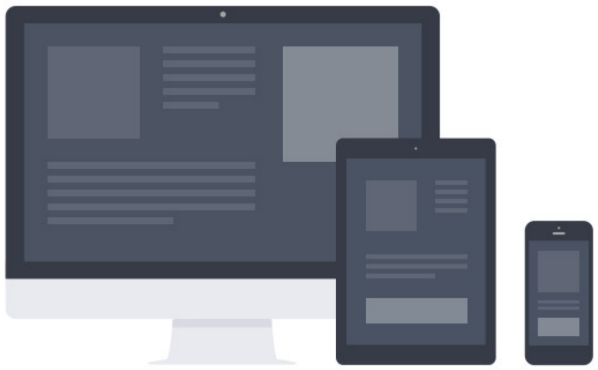
Key is flexibility - Stretch and compress your layouts to accommodate various heights and widths.

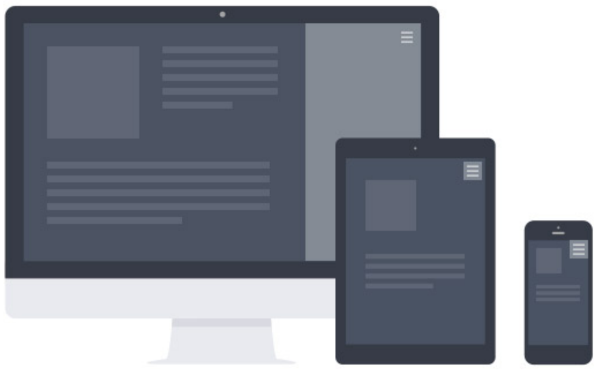
Optimize layouts - On larger devices, take advantage of extra screen real estate. Create compound views that combine multiple views to reveal more content and ease navigation.

Tiling/paneling – Tiling or paneling is a great way for your app to keep a fluid layout within a grid layout system. You can combine multiple panels into one compound view when a lot of horizontal screen real estate is available and split them up when less space is available.

- Stretch/compress - Adjust the column width of your panels to achieve a balanced layout in different orientations.
- Stack/wrap - Rearrange the panels on your screen to match the orientation (ie. going from a row of 2 column panels to a single column panel with 2 rows)
- Reveal/hide - When the device screen size changes or rotates, consider collapsing less prominent panels to only show the most important information.


Stretch/compress


Stack/wrap


Reveal/hide

Checklist

- Utilize resizable, rich, high-res graphics.
- To ensure an intuitive navigation experience, design well-defined, clear task flows with minimal navigation steps
- Provide enough room for user's fingers
- Plan how the content for your widget should adapt to different screen sizes.
- Make sure that your app consistently provides a balanced and aesthetically pleasing layout by adjusting its content to varying screen sizes and orientations.

Design For Different Platforms

Remember that iPhone experience is different than Android, not just in look, but also function, therefore we recommend that developers familiarize themselves with the design principles of both Android and iOS.

Android App Developers Guide - http://developer.android.com/design/index.html
iPhone Human Interface Guidelines - https://developer.apple.com/library/ios/design