



Red Hat Mobile Application Platform 4.7

Mobile Developer Guide

For Red Hat Mobile Application Platform 4.7

Red Hat Mobile Application Platform 4.7 Mobile Developer Guide

For Red Hat Mobile Application Platform 4.7

Legal Notice

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document provides guides for development of mobile Client Apps in Red Hat Mobile Application Platform 4.7.

Table of Contents

CHAPTER 1. ANDROID	4
1.1. DEPLOYING AN APP ON ANDROID	4
1.1.1. Browser Method	4
1.1.2. Dropbox Method	4
1.1.3. Android Tools Method	4
CHAPTER 2. IOS	5
2.1. DEPLOYING AN APP ON IOS	5
CHAPTER 3. FORMS	7
3.1. INTEGRATING FORMS INTO A CORDOVA APP	7
3.1.1. Working Example	7
3.1.1.1. Overview	7
3.1.1.2. Cordova App	7
3.1.1.2.1. Job Model	8
3.1.1.3. Cloud App	8
3.1.2. Creating A Working Example	8
3.1.3. Implementation Guide	9
3.1.4. Related Sections	12
CHAPTER 4. DEVELOPING CLIENT APPS	14
4.1. DEVELOPING AN IONIC APP USING RHMAP	14
4.1.1. Sample Project Overview	14
4.1.1.1. Client App	14
4.1.1.2. Cloud Code App	14
4.1.2. Create A New Ionic Hello World Project	14
4.1.3. Development Overview	15
4.1.3.1. Cloud Code App	15
4.1.3.2. Ionic Client App	15
4.1.3.2.1. fhconfig.json	15
4.1.3.2.2. \$fh.cloud	16
4.2. USING CORDOVA PLUG-INS	16
4.2.1. Supported Platforms	16
4.2.2. Adding Plug-ins to Apps	16
4.2.2.1. Specification	17
4.2.2.2. Default Plugins	18
4.2.3. Testing Apps in the Browser	19
4.3. USING SECURE KEYS IN YOUR APP	20
4.3.1. CUID and App Id	20
4.3.2. Key Persistence	20
4.3.3. Sample Code	22
4.4. DEBUGGING APPS	22
4.4.1. Studio console	22
4.4.2. Firebug	22
4.4.3. Web Inspector with Chrome	23
4.4.4. Safari / iOS 6+	23
4.4.5. Chrome / Android 4.0+	24
4.4.6. On-Device	27
4.4.6.1. Weinre	27
CHAPTER 5. PUBLISHING APPS	29
5.1. SUBMITTING AN APP TO GOOGLE PLAY	29

5.1.1. Prerequisites	29
5.1.2. Overview	29
5.1.3. Uploading an Application	29
5.1.4. Upload Assets	29
5.1.5. Listing Details	30
5.1.6. Publishing Options	31
5.1.7. Contact Information	31
5.1.8. Consent	32
5.2. SUBMITTING AN APP TO THE APPLE APP STORE	32
5.2.1. Creating a new Application	32
5.2.2. Submit Application for Sale	33
5.2.3. External Links	34
5.3. APP CREDENTIALS BUNDLES	34
5.3.1. Resources	34
5.3.1.1. All Platforms	34
5.3.1.2. Android Only	34
5.3.1.3. iOS Only	35
5.3.2. Apple Developer and Enterprise Accounts	35
5.3.2.1. Developer Account	36
5.3.2.2. Enterprise Account	36
5.3.2.3. iOS Certificate Renewal	36
5.4. USING 3RD PARTY MDM SERVICES	36
5.4.1. Uploading Client App binaries to MobileIron MDM repositories	36
5.4.1.1. Upload Client App binaries to MobileIron Cloud	36
5.4.1.2. Upload Client App binaries to MobileIron Core	37

CHAPTER 1. ANDROID

1.1. DEPLOYING AN APP ON ANDROID

There are many ways to install an **.apk** file onto an Android device or emulator. Some of these ways are detailed in the text below. If you do not have an Android device, use an Android emulator which is installed as part of Android SDK installation. See [Running Apps on the Android Emulator](#) for more information.

1.1.1. Browser Method

If you are using the [FHC command line tool](#) to generate your binary, the output will contain a URL value for example,

```
http://[your-studio-  
domain].redhatmobile.com/digman/A/B/C/android~2.2~50~MyApp.apk
```

This is the download URL of the binary. By navigating to this URL on the device browser, the binary can be downloaded and installed.

TIP

Using a URL shortener like [bit.ly](#) makes entering the download URL a lot easier

1.1.2. Dropbox Method

If you have a [Dropbox](#) account, there is a Dropbox app available in the Google Play Store. Placing the APK file in your Dropbox folder (on your PC/Mac) will make the file available through the Dropbox app, allowing you to open and install.

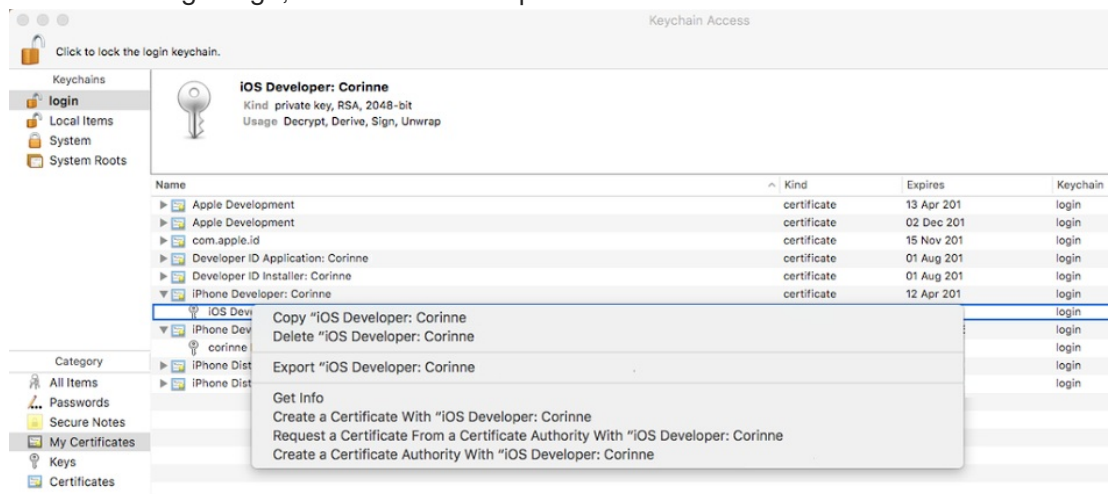
1.1.3. Android Tools Method

If you have the [android tools](#) installed, you can follow the [instructions](#) for sending the APK file to the device over a USB cable.

CHAPTER 2. IOS

2.1. DEPLOYING AN APP ON IOS

1. You must possess an iOS Development or Distribution Certificate, see [here](#) for more info.
2. Log on to the [iPhone Provisioning Portal](#) and use the test device's UDID to create a device.
3. Register the app's App ID and then create a Provisioning Profile, see [here](#) for more info.
4. Generate an iOS dev p12 file. A MAC is required to complete the following steps:
 - a. Open the Keychain tool
 - b. Navigate to "My Certificates" and locate your iOS Development or Distribution Certificate
 - i. In the following image, an iPhone Developer certificate has been selected



- c. Locate the Private Key and right-click on it. See the image above for reference
 - d. Select the option 'Export "iOS developer:"<Name of Developer>' and export this file as a p12 file
 - i. When prompted to enter a passphrase for the p12 file, enter a secure passphrase
 - ii. When prompted for a password for the keychain, enter the MAC password
 - A. The p12 will be stored in the MAC's keychain
5. Create a Credential's Bundle in the RHMAP Studio:
 - a. Login to the RHMAP Studio
 - b. In the 'Project' section, locate your iOS app and click on it
 - c. Locate the 'Credentials' tab, click on it and then click 'Create New Bundle'
 - d. Select the 'iOS' icon and enter data into the fields:
 - i. **Bundle name:** Enter a meaningful name
 - ii. **Private key:** Select the p12 file that was exported

- iii. **Certificate:** Select the iOS Development or Distribution Certificate
 - iv. **Provisioning Profile:** Select the Provisioning Profile created in the steps above
 - e. Finally, click 'Create Bundle' to create a Credential's Bundle
6. Use the Credential's Bundle to build the application using the RHMAP Studio. A QR code will be created.
7. There are two ways to install the app:
- a. Use the QR code to install the app on the test device
 - b. Download the binary, an **.ipa** file, from the RHMAP Studio and install the **.ipa** file using iTunes

CHAPTER 3. FORMS

3.1. INTEGRATING FORMS INTO A CORDOVA APP

Forms functionality can be integrated into existing apps. This guide demonstrates the integration on an example of a workforce management application — a supervisor assigns jobs, a worker receives the assignments on their mobile device, and sends back information about the job.

The procedures in this guide use a [Cordova app](#) and a [Cloud App](#). These apps can be imported into a project to demonstrate the working example described in this guide.

3.1.1. Working Example

3.1.1.1. Overview

The following requirements are set for the example application:

A supervisor creates a job for a worker to remove a fallen tree after a storm. The supervisor asks the worker for details of the job such as photos of the tree, location, comments, and the time started and finished. This can be achieved using forms apps.

The supervisor creates a job by:

- Selecting the form for the worker to complete.
- Selecting and filling out a form with the details for the job.
- Giving the job a unique ID.

The requirements for the application include:

- Jobs are created by an admin by filling out a creation form.
- New jobs are available to both admin and non-admin users. App users can view the completed creation form.
- App users can complete a job by filling in a separate completion form.
- The completion form must be visible to any app user to review.
- The Client App user can mark the job as either "In Progress" or "Complete" by saving the completion form as a draft.

3.1.1.2. Cordova App

The Client App is based on the following technologies:

- [Backbone](#): The Cordova app uses backbone models and views to manage the creation and update of jobs.
- [Handlebars](#): Used for view templating.
- [Bootstrap](#): Used for styling.
- [Font-Awesome](#): Used for icons.

The Cordova app is responsible for:

- Managing the listing of jobs in various states.
- Managing the rendering of any forms.
- Managing the submission and upload of any form submissions.
- Managing the creation of jobs containing form and submission data.

3.1.1.2.1. Job Model

The job model is a simple Backbone model describing a job.

The jobs collection is a collection of job models.

A custom URL is included for synchronizing jobs between the client and cloud. This custom URL is used to access RESTful `/jobs` endpoints on the Cloud App.

3.1.1.3. Cloud App

The [Cloud App](#) consists of RESTful endpoints (`/jobs`) for performing CRUD operations on job data using the `$fh.db API`.



NOTE

There is no visible logic in the Cloud App to deal with forms, because all cloud-based forms logic is contained in the [cloud APIs](#).

3.1.2. Creating A Working Example

1. Create a new project in the App Studio.
2. Import an app into your project. For example, the examples in this guide use the [Cordova app](#) and [Cloud App](#).
3. Create [forms](#) and [themes](#) for the project. Any created forms and themes associated with the project will now be visible in the Cordova app.
4. Optionally, add users and field codes to the project. For example:
 - Admin: Able to create and complete jobs.
 - User: Able to complete jobs.

A user has the **userId** and **userName** fields that are automatically added to a submission before rendering the related form. Add two text fields to any forms associated with this working example.

When the fields have been added, add two users to the **users** collection in your Data Browser.

Admin

```
{
  "userId": "admin",
  "userName": "<<Any Name>>"
}
```

```
  }
```

User

```
{
  "userId": "user",
  "userName": "<<Any Name>>"
}
```

3.1.3. Implementation Guide

Use the following to integrate forms functionality into an app.

1. Add Forms Initialization by adding the **\$fh.forms.init** function to the client. This initializes forms on the Client App to enable the usage of the **\$fh.forms Client API** in the rest of the app. The **\$fh.forms.init** function is part of the log in process for the app.
2. As an admin user, select a completion form. This specifies the form that needs to be completed in order to complete the job. List all of the forms available to the app using the **\$fh.forms.getForms** Client API function.



NOTE

The **\$fh.forms.getForms** Client API call only downloads a list of forms, it does not download the entire form definition for each form.

3. Download a form to the client using the **\$fh.forms.getForm** Client API. As forms are used in job creation, viewing job details, and completing jobs, this function is abstracted to a set of helper functions [here](#).

The **\$fh.forms.getForm** client API usage can be seen [here](#) as part of the **loadForm** function in **FormFunctions.js**.

4. Load a submission into your app. This process is illustrated using the **loadSubmission** function in the **FormFunctions.js** file. Forms are related to submissions, in that any data entered into a form is populated to a submission. However, a submission is validated against a form before being upload to the cloud.

There are three ways to create a submission:

- **From Local Memory:** Save a submission as a draft to local memory then edit later using the **saveDraft** function on the submission model. The implementation of this functionality is shown in the **loadLocalSubmission** function.
- **Download From Remote:** Download a submission from the cloud. For example, when the supervisor completes a form to describe the details of the job, the ID of the submission is saved to the job model. When the app user downloads the job model, they have access to the remote submission ID of the form submitted by the admin user. This remote submission ID is used to download the full submission definition from the cloud. The implementation of this functionality is shown in the **downloadSubmission** function.

**NOTE**

The form definition for the submission is contained in the submission downloaded from the cloud. This is because the form definition may have been edited between submissions.

**NOTE**

Downloaded submissions should not be edited on the client. They are intended for read-only access. Any attempt to submit a downloaded submission to the cloud will return an error.

- **Create A New Submission:** If there is no submission associated with a form, a new submission can be created. In this case, the submission is [created from a form model](#). This ensures that the submission is automatically related to the correct form.
5. Render the form into the view for editing by a user.
- There are two methods of rendering a form into an existing Cordova app:

- Rendering the form using the **\$fh.forms.backbone** API, which includes a backbone/bootstrap SDK (**\$fh.forms.backbone**), by downloading the [Appforms Backbone](#) file and include it as part of your Cordova app. In addition, the Cordova app must satisfy the following JavaScript and CSS dependencies:
 - Backbone
 - Bootstrap
 - Font-Awesome

The [CSS](#) and [JavaScript](#) dependencies are included in the example Cordova app.

The **FormViewSDK.js** file contains the Backbone SDK version of the form view. The Cordova app contains an option in the "Settings" tab to switch between the Backbone SDK and manual form rendering.

**NOTE**

The Backbone SDK is intended to speed up forms apps integration for Backbone/Bootstrap based Cordova apps. However, the **\$fh.forms** Client API will work with any Cordova app. The rendering of the form and managing the population of user data to a submission will be the responsibility of the developer.

- Rendering a form manually.

**NOTE**

Rendering a form to the user is the simplest method of completing a submission. However, field input values can be added to a submission from any source. The submission is still required to be valid against any field or page rules.

The **\$fh.forms** SDK does not depend on any framework, and can therefore be added to

any Cordova app. This app is based on Backbone and Bootstrap, however it is equally possible to use the **\$fh.forms** API with other javascript-based UI frameworks (for example, Angular).

A basic Bootstrap form is rendered based on the form definition. This form is defined in the **FormView.js** file. All of the rendering, submission input, and validation logic of the form is defined in the app using the **\$fh.forms** API and models.



NOTE

The manually rendered form is implemented for illustration purposes only. Only the text and number fields are manually implemented. However, all available form field types can be rendered using the **\$fh.forms.backbone** SDK.

The rendering logic for the custom form view is located in the **FormView.js** file. Here, you can see that the view handles all of the events related to [rendering](#) the form to the user.

In addition, the **FormView.js** file contains logic for:

- Validating field data when entered.
- Checking field and page rules.
- Populating data to a submission.
- Saving a submission as a draft.
- Submitting a form to the cloud.

The following steps illustrate how the Cordova app addresses these requirements when manually integrating the **\$fh.forms** SDK into a custom rendered form.

6. Define the validation parameters that restrict the data that can be entered into the field (for example, a text field can specify a minimum/maximum number of characters that can be entered into the field). Adding this functionality to the Client App reflects the restrictions of the field. To satisfy this requirement, the [validateInput](#) function is registered to the blur event of an input in the **FormView.js** file.



NOTE

Validation parameters influence whether a submission is valid. Even if field validation is not performed on the Client App, all submission fields will be validated before saving to the database.

7. Form apps include field and page rules. In the Studio, forms editors can create field rules to show and hide fields based on field input data and page rules to show and skip pages based on field input data.

This functionality is reflected in the implementation of the **\$fh.forms** API. By processing a submission using a rules engine, the submission can identify fields or pages that need to be shown or hidden.

This is implemented in the [checkRules](#) function in the **FormView.js** file.

**NOTE**

Field and page rules influence whether a submission is valid. Even if field and page rules are not checked on the Client App, the submission will be checked against all rules before saving to the database.

8. Add data to a submission model using the **addInputValue** function. The source of this data can either be the form rendered to the user, external data available to the app, or a mixture of both.

- From a rendered form: In this case, a form is rendered for the user to input data using the **\$fh.forms.backbone** SDK or by manually rendering a form.
When manually integrating the **\$fh.forms** API into a custom rendered forms, it is necessary to handle the migration of data from the view to the submission model.

This is illustrated by the [saveFieldInputsToSubmission](#) function in the **FormView.js** file.

- From an external source using field codes: You can add field codes to form fields to uniquely identify a field within a form. This field code can relate to an external data source (for example, a header in a CSV file). Using this functionality, it is possible to import external data into a form submission.

This functionality is demonstrated in the example Cordova app by the [addSubmissionData](#) function. In this example, a user has **userId** and **userName** fields. If a form contains fields with fields codes **userId** and **userName**, these fields will be populated with the data from the User model.

**NOTE**

Field codes must be unique within a form. However, the same field code can be present in multiple forms.

9. Save a submission as a draft. This functionality is illustrated by the [saveDraft](#) function in the **FormView.js** file.
10. Having added validation and rules functionality to the form, we can now submit valid submissions to the cloud for viewing/editing on the submission editor.
The form view listens for submission-related events ([validationerror](#), [queued](#), [progress](#), [error](#), [submitted](#)) emitted by the submission model as the data is being processed and uploaded.

The submission process has two distinct steps:

- **Submit**: Calling the submit function on a submission model validates the submission against the local form definition and changes the submission status to pending.
- **Upload**: Calling the upload function on a submission model will queue the submission for upload to the forms database.

3.1.4. Related Sections

- [\\$fh.forms Client API](#)
- [\\$fh.db Cloud API](#)
- [Creating A Form](#)

- [Creating A Theme](#)
- [Creating A Forms Project](#)

CHAPTER 4. DEVELOPING CLIENT APPS

4.1. DEVELOPING AN IONIC APP USING RHMAP

Overview

This guide will introduce you to using the RHMAP Javascript SDK within a HTML5 Ionic Cordova App.

This guide includes the steps necessary to create a new Ionic Hello World Project and highlights the code necessary to interact with a Cloud Code App.

Requirements

- RHMAP Account
- Knowledge of HTML, JavaScript, [Ionic](#) and Node.js
- GitHub User Account
- [FHC](#) installed.

The [Ionic Documentation](#) contains all of the information needed to start developing Ionic Apps.

4.1.1. Sample Project Overview

The example project is a simple project containing one Client App and one Cloud App.

4.1.1.1. Client App

The Client App is a simple Ionic App that includes the RHMAP Javascript SDK. The Client App asks the user to enter their name into a text box and click a **Say Hello From The Cloud** button. The Client App then uses the `$fh.cloud` function to send the text entered to the Cloud Code App.

4.1.1.2. Cloud Code App

The Cloud Code App exposes a **hello** endpoint to receive a request from the Client App, change the text sent to add **Hello** and return the response to the client.

4.1.2. Create A New Ionic Hello World Project

Use the following steps to create a new Ionic Hello World Project.

1. Log into the Studio.
2. Select the **Projects** tab located at top-left of the screen.
3. Select the **+ New Project** button located at the top of the screen. You will then see a list of Project Templates.
4. Select the **Hello World Project** Template and give the new project a name.
5. Click the **Create** button. The new project will now be created.
6. When the project has been created, select the **Finish** button.

You now have a new Project containing an Ionic Client App and a Cloud Code App that it will communicate with.

To Preview the app:

1. Select the **Apps, Cloud Apps & Services** tab.
2. Select the **Client App** under the **Apps** section. This will take you to the Details Screen.
3. The **App Preview** contains a working version of the Ionic Client App that will communicate with the Cloud Code App.

4.1.3. Development Overview

This section will highlight the code necessary for the example solution to work correctly.

4.1.3.1. Cloud Code App

First, let's consider the Cloud Code App. In this example, we want the Cloud Code App to receive a request from the Client App, change the **hello** parameter and respond to the Client App using a JSON object containing the following parameters:

```
{
  msg: "Hello <<hello parameter sent by the client app>>"
}
```

To implement this functionality in the Cloud Code App:

1. In the [application.js](#) file, a new **/hello** route is added which requires a **hello.js** file located in the **lib** directory.
2. The [hello.js](#) file creates two routes. Both routes perform the same operation of changing the **hello** parameter.
 - A **GET** request where the **hello** parameter is appended as a query string.
 - A **POST** request where the **hello** parameter is sent in the body of a POST request.



NOTE

This Cloud Code App is completely independent of the Ionic Client App. The Cloud Code App can be shared between any number of Client Apps within a project.

4.1.3.2. Ionic Client App

Having created the **/hello** endpoint in the Cloud Code App, we now proceed to examine the functionality added to the Ionic Client App to allow it to send requests to the **/hello** endpoint exposed in the Cloud Code App.

The Client App is a simple Ionic App with a single input that accepts some text and a single button that sends the input to the cloud and displays the result to the user.

4.1.3.2.1. fhconfig.json

The Client App contains a [fhconfig.json](#) file. This file contains the information needed for the RHMAP Javascript SDK to communicate with the Cloud App.



NOTE

All HTML5 Client Apps must contain a **fhconfig.json** file to use the \$fh Client API functions. This file is automatically populated with the required information when the app is created in the Studio.

4.1.3.2.2. \$fh.cloud

In this example, the **\$fh.cloud** Client API function is used to send requests to the **hello** endpoint in the Cloud Code App.

The **\$fh.cloud** function is located in the [fhcloud.js](#) file. Here, the \$fh.cloud function is exposed as a reusable service for the **MainCtrl** Controller to use.

There is a single controller in the Ionic App called [MainCtrl](#). This controller is responsible for

1. Accepting the input from the user from the [example.html](#) view.
2. Using the **fhcloud** service to call the **hello** endpoint in the Cloud Code App.
3. Processing the response from the Cloud Code App using the **success** or **error** functions, depending on whether the \$fh.cloud call was successful.



NOTE

In this case, the Client App is using a **GET** request type. As the Cloud Code App exposes both a **GET** and **POST** version of the **hello** endpoint, a **POST** request type will also work. This is especially useful when dealing with RESTful applications.

4.2. USING CORDOVA PLUG-INS

Cordova plug-ins provide a platform-independent JavaScript interface to mobile device capabilities in hybrid mobile apps. There are several official Apache plug-ins for basic device functions such as storage, camera, or geolocation, and other third-party plug-ins.

The official registry and distribution channel for Cordova plug-ins is npm and the plug-in ID corresponds to the npm package ID. The [Cordova Plugins](#) page contains the official plug-in list and acts as a filter for npm packages with the keyword **ecosystem:cordova**.

4.2.1. Supported Platforms

You can use Cordova plug-ins with all platforms supported by RHMAP. However, not all Cordova plug-ins support all platforms. Platforms supported by a plug-in are specified in a plug-in's **package.json** file in the **cordova.platforms** object, or on the Cordova Plug-ins search page.

4.2.2. Adding Plug-ins to Apps

When developing Cordova applications, plug-ins are added using **cordova plugin add**, which downloads the plug-in from the repository, creates the necessary folder structure, and adds an entry in the **config.xml** file. See [Platforms and Plugins Version Management](#) in the official Cordova

documentation for more information.

In RHMAP, however, plug-ins can also be declared in an RHMAP-specific JSON file **config.json** and makes the apps future-proof in case the plug-in specification format changes.

```
{
  "plugins": [
    {
      "id": "cordova-plugin-device",
      "version": "latest"
    },
    {
      "id": "cordova-plugin-geolocation",
      "version": "1.0.1"
    },
    {
      "id": "cordova-labs-local-webserver",
      "url": "https://github.com/apache/cordova-plugins#master:local-
webserver",
      "version": "2.3.1",
      "preferences": {
        "CordovaLocalWebServerStartOnSimulator": "false"
      }
    }
  ]
}
```

Three plug-ins are specified in the above example: the Device plug-in and Geolocation plug-in, which are available through npm, and the Local WebServer plug-in which is available only from the indicated Github repository.

4.2.2.1. Specification

The **config.json** file must be located in the **www** folder of a Cordova app.

The file must contain a key called **plugins**, the value of which is an array of JSON objects, which can define the following properties:

- **id** (Required)
This is the globally unique ID of the Cordova plug-in, which corresponds to its npm package ID. It can also be found in the plug-in's **plugin.xml** file, as described in the [Cordova Plugin Specification](#).
- **version** (Required)
The version of the plug-in to use. Corresponds to the npm package version. For plug-ins distributed through Git only, you can find the version of a plug-in in its **plugin.xml**.

For plug-ins distributed through npm, you can also use the **latest** value to always use the latest available version. We strongly advise you to use a specific version that is proved to work with your app, since a plug-in upgrade could break backward compatibility.

- **url**
The URL of a public Git repository, containing a valid Cordova plugin (contains a **plugin.xml** file).

The provided value for this field is used to download the plug-in, regardless of the values of the **id** and **version** fields. If this field is not provided, the plug-in will be downloaded from npm with the values of **id** and **version** fields.

You can also specify a particular Git ref and a path to the plug-in within the repository, as described in [official Cordova CLI documentation](#).

- A Git ref object can be specified by appending **#<git-ref>** in the URL. We strongly recommend using a tag or other stable ref that is tested as working with your app. Using **master** as the ref could result in the plug-in code changing on every build and potentially breaking your application.
- Path to the directory containing the plug-in can be specified with **:<path>** in the URL.

For example, if we want to get a plug-in that resides in the **plugin** subdirectory in the **release** branch of the repository, the URL should have the following format:

```
https://example.com/example.git#release:plugin
```

After the plug-in is downloaded, its **plugin.xml** file will be parsed to make sure the plug-in ID matches the **id** field specified in the **config.json**.

- **preferences**
To provide plug-in configuration, use key-value pairs in the **preferences** object, like **CordovaLocalWebServerStartOnSimulator** in the example above.
- **variables**
Some plug-ins use variables in **plugin.xml** to parameterize string values. You can provide values for variables as key-value pairs in this field. See [official documentation](#) for more information on variables.

4.2.2.2. Default Plugins

Official plug-ins

- cordova-plugin-device (1.1.2)
- cordova-plugin-network-information (1.2.1)
- cordova-plugin-battery-status (1.1.2)
- cordova-plugin-device-motion (1.2.1)
- cordova-plugin-device-orientation (1.0.3)
- cordova-plugin-geolocation (2.2.0)
- cordova-plugin-file (4.2.0)
- cordova-plugin-camera (2.2.0)
- cordova-plugin-media (2.3.0)
- cordova-plugin-media-capture (1.3.0)

- cordova-plugin-file-transfer (1.5.1)
- cordova-plugin-dialogs (1.2.1)
- cordova-plugin-vibration (2.1.1)
- cordova-plugin-contacts (2.1.0)
- cordova-plugin-globalization (1.0.3)
- cordova-plugin-inappbrowser (1.4.0)
- cordova-plugin-console (1.0.3)
- cordova-plugin-whitelist (1.2.2)
- cordova-plugin-splashscreen (3.2.2)

Optional

- cordova-sms-plugin (v0.1.9)
- com.arnia.plugins.smsbuilder (0.1.1)
- cordova-plugin-statusbar (2.1.3)

Custom RHMAP plug-ins:

- [com.feedhenry.plugins.apis](#) (0.0.6)
- [com.feedhenry.plugins.apkdownloader](#) (0.0.1)
- [com.feedhenry.plugin.device](#) (0.0.2)
- [com.feedhenry.plugins.ftputil](#)

4.2.3. Testing Apps in the Browser

If a plug-in is specified in the `config.json` file, the JavaScript object of the plug-in won't be available until it is built and installed on the device. So if you reference the plug-in's JavaScript object in your app, and try to load your app in *App Preview* in the Studio, you may get *object undefined* errors.

To solve this, use defensive checking when calling the plug-in APIs. For example, the following call to the Cordova camera API would result in an "object undefined" error:

```
navigator.camera.getPicture(success, fail, opts);
```

However, checking whether the API is defined lets you handle the error gracefully:

```
if(typeof navigator.camera !== "undefined"){
  navigator.camera.getPicture(success, fail, opts);
} else {
  //fail gracefully
  fail();
}
```

4.3. USING SECURE KEYS IN YOUR APP

`$fh.sec` APIs provides the functionality to generate keys and data encryption/decryption. However, after the keys are generated, you may need to save them somewhere for future usage. For example, you have some data that needs to be encrypted with a secret key and saved on the device. Next time, when the app starts again, you need to get the same secret key and decrypt the data.

The best practice to achieve this is to save the keys on the cloud side, and associate the keys with the client using the client unique id (CUID) and app id.

4.3.1. CUID and App Id

Both the CUID and app id are sent by the client SDK in every `$fh.act` request. They are accessible via a special JSON object called `__fh`. The CUID is unique for each device and remain unchanged even if the app is deleted and re-installed. The app id is generated when the app is created on the platform and remain unchanged. You can use the following code to access them in the cloud code:

```
getKeyId = function(params){
  var cuid = params.__fh.cuid;
  var appid = params.__fh.appid;
  var keyid = cuid + "_" + appid;
  return keyid;
}

exports.getKey = function(params, callback){
  var keyid = getKeyId(params);
  //get a key using this keyid
  ....
}
```

4.3.2. Key Persistence

You can use whatever persistent mechanism you like to save the keys in the cloud. One recommended approach is to use `$fh.db`. Here is some example code to show how to save and retrieve keys using `$fh.db`:

```
//read a key using $fh.db
var getKey = function(params, cb){
  if(typeof $fh !== "undefined" && $fh.db){
    $fh.db({
      act:'list',
      'type': 'securityKeys',
      eq: {
        "id": getKeyId(params), //The id is generated using the above
example code
        "keyType": params.type
      }
    }, function(err, data){
      if(err) return cb(err);
      if(data.count > 0){
        return cb(undefined, data.list[0].fields.keyValue);
      } else {
        return cb(undefined, undefined);
      }
    });
  }
}
```



```

    });
  } else {
    console.log("$fh.db not defined");
    cb("$fh.db not defined");
  }
}

//save a key using $fh.db
var saveKey = function(params, cb){
  if(typeof $fh !== "undefined" && $fh.db){
    //first check if a key with the same id and type already exists
    $fh.db({
      act:'list',
      'type': 'securityKeys',
      eq: {
        "id": getKeyId(params),
        "keyType": params.type
      }
    }, function(err, data){
      if(err) return cb(err);
      //a key with the same id and type already exists, update it
      if(data.count > 0){
        $fh.db({
          'act':'update',
          'type': 'securityKeys',
          'guid': data.list[0].guid,
          'fields' : {
            'id': getKeyId(params),
            'keyType': params.type,
            'keyValue' : params.value
          }
        }, function(err, result){
          if(err) return cb(err);
          return cb(undefined, result);
        })
      } else {
        //a key with the same id and type is not found, create it
        $fh.db({
          'act': 'create',
          'type': 'securityKeys',
          'fields': {
            'id' : getKeyId(params),
            'keyType': params.type,
            'keyValue': params.value
          }
        }, function(err, result){
          if(err) return cb(err);
          return cb(undefined, result);
        })
      }
    });
  } else {
    console.log("$fh.db not defined");
    cb("$fh.db not defined");
  }
}

```

4.3.3. Sample Code

A reference application has been created which fully demonstrates how to use **\$fh.sec** APIs. The code for this application is available on GitHub: <https://github.com/feedhenry-training/fh-security-demo-app>.

4.4. DEBUGGING APPS

Debugging is an essential part of app development. With web applications this can be as simple as using the **alert()** function to show the value of a variable at a certain point in the code. For browsers that have a console, the **console.log** function can be used to log this kind of information passively. More advanced forms of debugging web applications include inspecting the state of the DOM, breakpointing JavaScript code and stepping through it, or updating the DOM and the associated styles on the fly.

This page gives an explanation of the debugging tools that can be used while developing cross platform apps. These tools can be used in the Studio or on device.

For more information about how to prepare an app for debugging and how to debug on iOS and Android, see this [blog](#).

4.4.1. Studio console

Most browsers support the console logger. Various log levels can be called with a message for example,

```
console.log(message);  
console.error(message);
```

To debug this console output, you will need to open the relevant web debugging tools in your browser.

For debugging cloud code, simply use **console.log()** and **console.error()**. The corresponding log files can be viewed in the Studio under the 'Logs' section, or using FHC.

```
console.log('this goes to stdout.log');  
console.error('this goes to stderr.log');
```

4.4.2. Firebug

If your browser of choice for development is Firefox, the [Firebug](#) tool makes debugging very easy. Firebug is an add-on for Firefox that is quite active and has many updates to it. Here are some of the things that can be done using Firebug:

- view console output
- view resource requests
- debug script execution
- dynamically run code, even when breakpoint debugging
- view/update the DOM
- view/update styles
- view/update local storage (DB)

Using these features, debugging an app is made very easy. Getting an app working without Firebug showing any errors or problems gives the app a good chance of working cross platform.

4.4.3. Web Inspector with Chrome

Web Inspector is very similar to Firebug. It offers more or less the same features as Firebug, but has the advantage of being included with WebKit browsers out of the box. This means the Web Inspector can be used with Google Chrome and Safari. The tool is enabled by default with Chrome and can be started by context clicking any object on a web page, and selecting **Inspect Element**.

4.4.4. Safari / iOS 6+



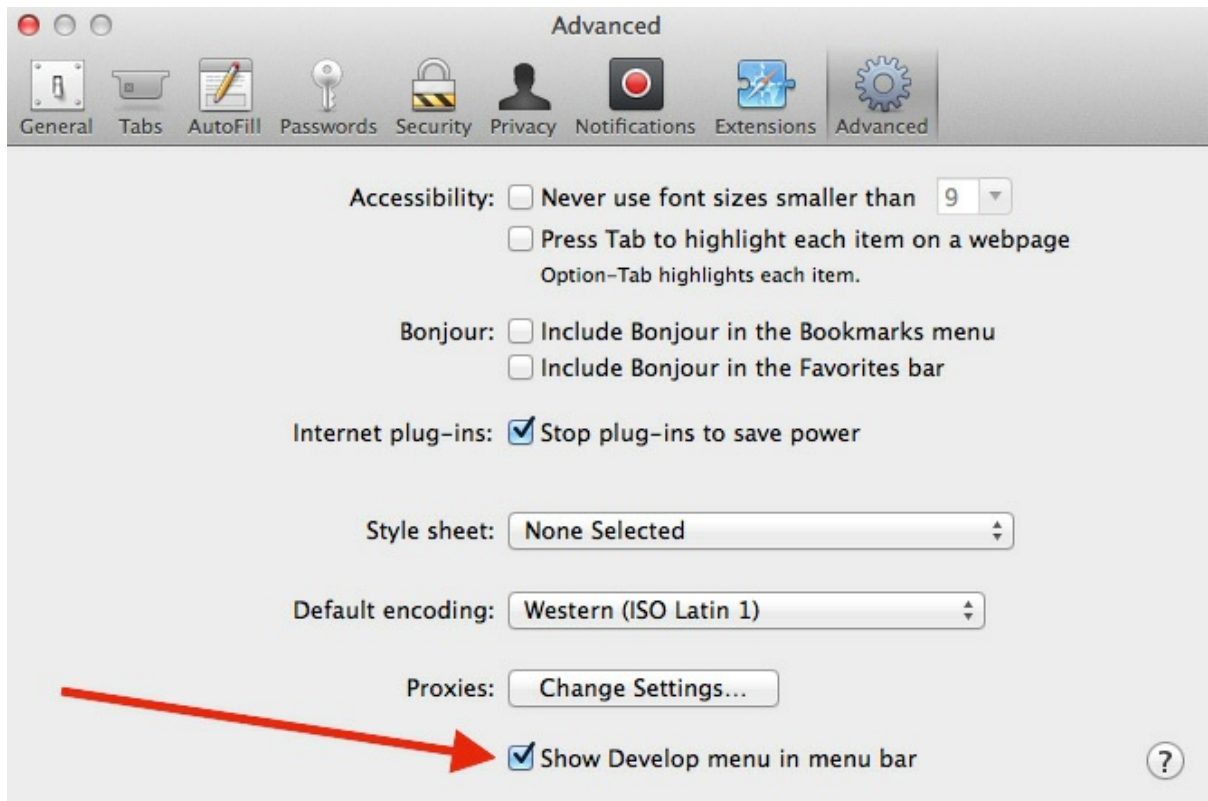
NOTE

Remote debugging on iOS with Safari can only be enabled for applications built with a Development provisioning profile. Ad Hoc and Distribution apps cannot be debugged this way.

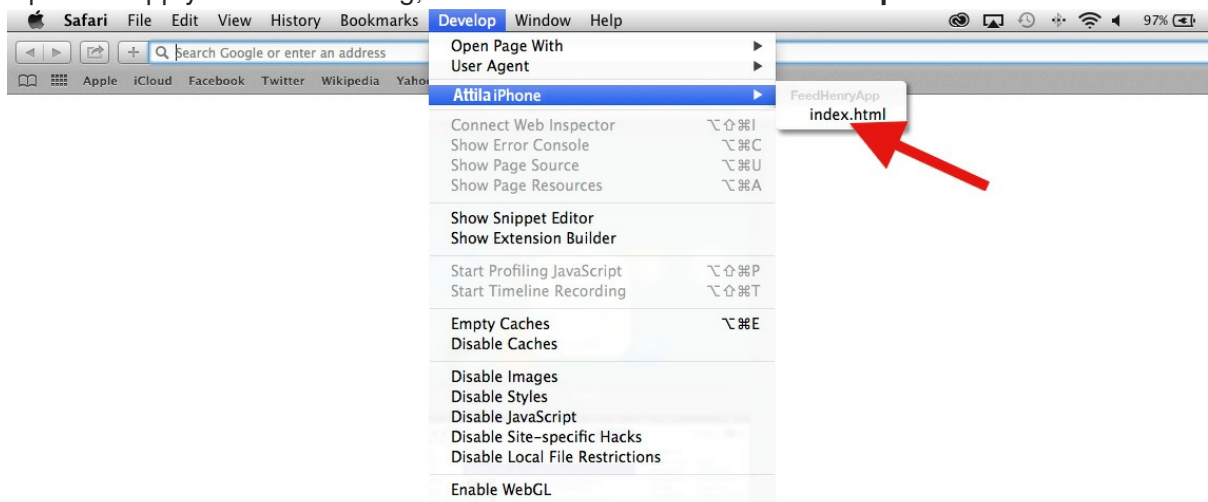
1. Go to the **Settings app** on your iOS device.
2. Navigate to **Safari** → **Advanced**, and then toggle on the **Web Inspector** switch.



3. In desktop Safari, go to **Safari** → **Preferences**, select **Advanced**, then check the **Show Develop menu in menu bar** check box.



4. Connect your iOS device to your development machine via USB.
5. Open the app you want to debug, it will be available in Safari's **Develop** menu.



6. The **Develop** menu has additional tools such as the **User Agent** switcher. This allows the browser to pretend to be a different browser for example, Mobile Safari. This feature can be useful if developing a mobile Internet version of an app.

4.4.5. Chrome / Android 4.0+



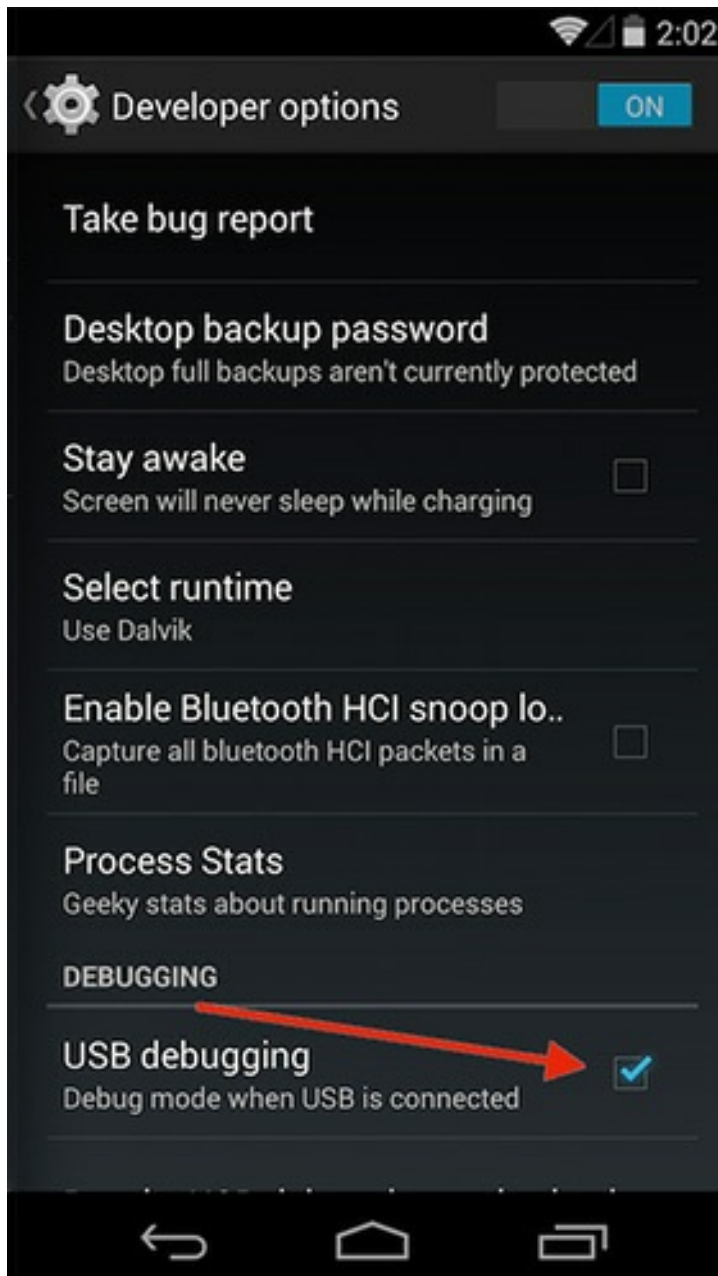
NOTE

Remote debugging on Android with Chrome can only be enabled for applications that flagged debuggable. If you are using Cordova 3.3 or newer, add **android:debuggable="true"** inside the **<application>** element in the **AndroidManifest.xml** file. If you are using Cordova 3.2 or older version, you have to enable WebView debugging.

1. Enable USB debugging on your Android device On Android 4.2 and newer versions, the developer option is hidden. To make it available, navigate to **Settings** → **About phone**, and tap the **Build number** seven times.



2. Go back to the previous screen to find **Developer options**, then check the **USB debugging** check box. Choosing the **Stay awake** option is also recommended.



3. In the desktop Chrome browser, in the menu on the upper-right corner select **Tools** → **Inspect Devices**
4. Check **Discover USB devices**, if it is not selected.
5. Connect your Android device via USB
6. If it is the first time you attached this device for developing, you may see an alert on device requesting permission for USB debugging from your development machine. To avoid appearing this alert every time you debug, check **Always allow from this computer**, then tap OK.
7. To debug your app, you need to run it on device.
8. In Chrome, click the **Inspect** link to open the DevTools.



4.4.6. On-Device

Debugging on device is still in an early stage compared to debugging in a desktop browser. Some platforms, such as iOS, make the console available and show any javascript errors when Developer mode is enabled, but it is nowhere near as fully featured as Firebug or Web Inspector.

4.4.6.1. Weinre

[Weinre](#) is probably the best on-device debugging tool available at the moment. It officially only supports webkit browsers. This means it will work for Android and iOS. The full list and version numbers are available on its site.

Weinre works by setting up a remote debugging session from the app on the device to the Weinre server. Weinre runs a web server which can be used to access the Web Inspector, and remotely debug the app. At the time of writing, the features of this Web Inspector session allow debugging the DOM and updating it. The remote console is also shown, and the developer can dynamically run code in the app from the console.

To enable Weinre debugging in an application, there are a few steps.

1. Get the Weinre jar up and running on a machine that the device can connect to. There are 2 ways of doing this:
 - Run the jar file on a machine accessible over the Internet
 - Run the jar on a machine on the same network as the device, that is, device connected via WiFi to the same router/access point as the developers machine.

Note that when running the jar from the command line, it is advised to use the value **-all-** for the **boundHost** so that it is listening on all interfaces.

2. Add a script to the HTML of the app before building and deploying it to the device. According to the documentation, the script include will look something like this.

```
<script src="http://some.server.xyz/target/target-script-min.js#anonymous"></script>
```

The address used, **some.server.xyz**, must match the address (ip address should work too) of the machine that is running the Weinre server.

3. Deploy the app to the device and launch it. Opening the web page of the Weinre server (on the developer machine) should present a link to the debug user interface. This link opens up the Web Inspector and allows remote debugging of the app.

As these instructions are for a third party tool, it is best to check with the official site for any updates around this setup process.

CHAPTER 5. PUBLISHING APPS

5.1. SUBMITTING AN APP TO GOOGLE PLAY

5.1.1. Prerequisites

- A 'Release' app (APK file) is built either from the Platform or using Android Studio
- The developer has a high resolution application icon file
 - 512x512 JPEG or 24 bit PNG (no Alpha)
- The developer has at least one screenshot of the app
 - 320x480 or 480x854 JPEG or 24 bit PNG (no Alpha) format



NOTE

Screenshots must be full bleed, have no border and landscape thumbnails will be cropped. All measurements WxH for example, 320wx480h

5.1.2. Overview

There are a number of steps for publishing an app to Google Play, all of which are done with a web browser. They are:

- Upload an Application
 - Upload Assets
 - Listing Details
 - Publishing Options
 - Contact Information
 - Consent

5.1.3. Uploading an Application

- Log into Google Play at <https://play.google.com/apps> using your Developer Login
 - You will be presented with a list of apps you already have in Google Play
 - Click on the Upload Application button
 - You will now be brought to the **Upload an Application** Screen

5.1.4. Upload Assets

- Draft application APK file
 - Upload your APK file
- Screenshots

- Upload your screenshot(s) -- JPEG or 24-bit PNG (Alpha Transparency is unsupported)
- High Resolution Application Icon
 - Upload your App Icon — JPEG or 24-bit PNG (Alpha Transparency is unsupported)
- Promotional Graphic
 - This is optional to be used in Google Play on devices with 1.6 or higher
- Feature Graphic
 - This is optional
- Promotional Video
 - This is optional to supply a YouTube URL to your promotional video
- Marketing Opt-out
 - There is also the option to opt out of Google marketing/ promoting your app

5.1.5. Listing Details

- Language
 - Select the Language for your app
- Title
 - The name for your application as it will appear in Google Play
- Description
 - Enter the description of your to appear in Google Play
- Recent Changes
 - Enter the changes as you are uploading updates of your application
- Promo text
 - This will be used in conjunction with the Promotional Graphic
- Application Type
 - Select the type of application it is "Applications" / "Games"
- Category
 - Select the category you want the app to be displayed in based on choice of application type
 - **Applications**
 - Comics
 - Communication
 - Entertainment

- Finance
- Health
- Lifestyle
- Multimedia
- News & Weather
- Productivity
- Reference
- Shopping
- Social
- Sports
- Themes
- Tools
- Travel
- Demo
- Software libraries
- **Games**
 - Arcade & Action
 - Brain & Puzzle
 - Cards & Casino
 - Casual
- Price
 - Price is automatically set to **Free**. If you want to charge for apps you must set up a merchant account at Google Checkout.

5.1.6. Publishing Options

- Copy Protection
 - Protect apps being copied from the device - **Soon to be deprecated and superceded by the Licensing Service**
- Locations
 - Select Google Play markets you want the app to appear in

5.1.7. Contact Information

- Website
 - Your website URL
- Email
 - Your Email address
- Phone
 - Your phone number

5.1.8. Consent

You must verify that the application meets the Android Content Guidelines and that your app may be subjected to US laws regardless of your location or nationality and that, as such, your application is authorised for export from the US under these laws.

Finally, You must choose to Save what you have entered, delete what you have entered or Publish the application (This will publish the app and make it available in Google Play)

5.2. SUBMITTING AN APP TO THE APPLE APP STORE

Prerequisites

This guide assumes that a number of things have already been done by the developer. They are:

- A 'Release' App has been built either from the Platform or using xCode
- The developer has 2 icon files for the application
 - 57x57 PNG format
 - 512x512 PNG format
- The developer has at least one screenshot of the app 320x480 PNG format

Overview

There are a number of steps for publishing an app to the iPhone App Store, all of which are done with a web browser. They are:

- Create a new Application
- Set up any in-app purchasing
- Submit Application for Sale

5.2.1. Creating a new Application

1. Log into iTunes Connect at <http://itunesconnect.apple.com> using your Developer Login (that is, email address)
 - a. Click on Manage Your Applications
 - b. Click on Add New Application

- c. Select a choice for the "Export Laws and Encryption" question
2. Overview Tab
 - a. Fill in the Overview section using your app details
 - b. Select a choice for "Restrict this binary to a specific platform" question
 - c. Enter a Version Number for example, 1.0
 - d. Enter a unique value for the SKU Number: for example, the current date and time
 - e. Click the (blue) Continue button at the end of the screen.
3. Ratings Tab
 - a. Select the appropriate check boxes for you application
 - b. Click (blue) Save Changes button
4. Upload Tab
 - a. Click the Upload application binary later option
 - b. Upload the icons and screen shots mentioned in the [Prerequisites](#) (Large Icon and Primary Screen Shot are mandatory). *TIP: When adding additional screen shots, you may like to select them in reverse order so that they appear in the correct order when added by the uploader (which seems to add/upload them in the opposite order that you selected them).*
 - c. Click the (blue) Continue button
5. Localization Tab
 - a. Choose the appropriate language for you app
 - b. Click the Continue button
6. Pricing Tab
 - a. Choose the Availability Date (Start Date) of your app for example, today
 - b. Choose the appropriate Price Tier
 - c. Select the stores you want your app to be available in and continue
7. Review Tab
 - a. Choose the appropriate review store (for example, the country your company resides in)
 - b. Click the (black) Submit Application button.

5.2.2. Submit Application for Sale

1. Upload Application Binary
 - a. Once logged into iTunes Connect, click Manager your Applications
 - b. Click the application you want to submit for approval by Apple (Note the current status should be orange, indicating the Application binary is yet to be uploaded)

- c. Click the Upload Binary button (The maximum binary size accepted is 64MB)
 - d. Click Choose File and locate your binary zip file mentioned in the [Prerequisites](#).
 - e. Click Upload File
 - f. Click the Save Changes button in the bottom right
2. Confirm Review App Store
 - a. Click Manager your Applications
 - b. Click your application that was submitted for approval
 - c. Click the Edit Information button and open the Review Tab
 - d. Verify that the Review App Store is set to the country selected while creating the application (necessary due to a suspected bug in iTunes Connect)
 - e. Click Done and your application is now ready for approval by Apple

5.2.3. External Links

[In depth information on managing apps](#)

5.3. APP CREDENTIALS BUNDLES

A Credentials Bundle consists of a number of resources needed to perform a particular build. Here, the different resources are listed, along with a brief explanation of their purpose.

5.3.1. Resources

When performing build operations, a Credentials bundle can sometimes be required (depending on the build). A Credential bundle is a combination of resources, such as certificates, provisioning profiles, and private keys, necessary for performing specific types of builds, be it a development build, distribution build, debug build etc. Depending on both the platform, and the build type, different resources will be grouped together to constitute a bundle.

Listed below is a breakdown of resources that can be added to a Credentials Bundle, along with a brief description of what they are used for.

5.3.1.1. All Platforms

- Private Key
 - This is a file whose contents are known only to the owner. During the app building process, the app is digitally signed using this key. This means the developers digital signature is left on the App, allowing the App to be tied back to the developer.

5.3.1.2. Android Only

- Android Distribution Certificate - Used to build Apps for upload to the Google Play Store. This certificate is used to identify you as the developer upon upload to the market.
- Signing key

1. Use **keytool** to create a signing key:

```
keytool -genkey -v -keystore redhat.keystore -alias rhmap -keyalg
RSA -keysize 2048 -validity 10000
```

2. Export the java keystore key into pkcs#12 format:

```
keytool -importkeystore -srckeystore redhat.keystore -
destkeystore rhmap.p12 -deststoretype PKCS12 -srcalias rhmap
```

3. Extract the Distribution Certificate:

```
openssl pkcs12 -in rhmap.p12 -nokeys -out rhmap-cert.pem
```

4. Extract the Private Key

```
openssl pkcs12 -in rhmap.p12 -nodes -nocerts -out rhmap-key.pem
```

where:

- Private Key is **rhmap-key.pem**
- Certificate is **rhmap-cert.pem**

5.3.1.3. iOS Only

- iOS Development Certificate
 - Used to run an iOS App on devices during development.
- iOS Distribution Certificate
 - Used for submitting your iOS App to the App Store, and for distributing the App for On-Device testing. This is also used to identify you as the developer.



WARNING

If you encounter this problem, refer to the Red Hat Knowledge Base article [Swift-based iOS application crashes upon startup when signed using an enterprise distribution certificate without Organisational Unit field](#) for detailed instructions on how to resolve the problem.

- Provisioning Profile
 - Necessary in order to install development applications on iOS devices.

5.3.2. Apple Developer and Enterprise Accounts

In order to publish an app in the Apple App Store you must have an active apple account (developer or enterprise). This account will need to be renewed annually in order for associated apps to continue to be available in the App Store.

5.3.2.1. Developer Account

A developer account is used create an iOS distribution certificates used to publish apps to the apple App Store.

When a distribution certificate expires, if the iOS Developer account is still active, existing apps on the App Store will not be affected, they will continue be available within the App Store and apps already on device will continue to function as expected.

5.3.2.2. Enterprise Account

An enterprise account is used to create (in-house) distribution certificates which are needed to publish apps to the RHMAP app store or customer in-house MDM.

When an existing in-house certificate expires all apps built with that certificate will not run and further installs of this version of the app will not be possible.

The app will need to be rebuilt, signed with a new certificate, republished to the relevant store and then re-downloaded by all users.

5.3.2.3. iOS Certificate Renewal

Apple enterprise and developer certificates must be recreated every three years.

The customer email address associated with the developer account will receive advance notification of the impending renewal requirement.

While Red Hat might have been involved in assisting a customer with the initial setup of an Apple developer or enterprise account the ownership and responsibility for the apple account and the certificates created remains with the customer.

Upon receiving the certificate expiry notification it is recommended that the customer proactively renew the certificate in order to avoid interruption to their apps availability.

5.4. USING 3RD PARTY MDM SERVICES

5.4.1. Uploading Client App binaries to MobileIron MDM repositories

Include these scripts in your Jenkins pipeline to automatically upload Client App binaries to MobileIron.

5.4.1.1. Upload Client App binaries to MobileIron Cloud

Use [cloud_uploadapp.sh](#) to upload apps to MobileIron Cloud.

To use this script as part of a Jenkins pipeline, include it in the Client App repo, and update the Jenkinsfile to invoke the script after the build by adding the following stage:

```
stage("Upload To MobileIron Cloud"){
    def host = '<your MobileIron Cloud host here>'
    def username = '<your MobileIron Cloud username here>'
```



```

def password = '<your MobileIron Cloud password here>'
//only required if the file is not executable
sh (
    script: "chmod +x ./cloud_uploadapp.sh"
)
sh (
    //sample for upload for an Android app
    script: "./cloud_uploadapp.sh ${host} ${username} ${password} ANDROID
./app/build/outputs/apk/app-debug.apk"

    //it will be like this for uploading an iOS app
    //script: "./uploadapp.sh ${host} ${username} ${password} IOS
./helloworld-ios/build/Debug-iphoneos/myapp.ipa"
)
}

```

5.4.1.2. Upload Client App binaries to MobileIron Core

Use [core_uploadapp.sh](#) to upload apps to MobileIron Core. The API for uploading apps is different from MobileIron Cloud.

To use this script as part of a Jenkins pipeline, include it in the Client App repo, and update the Jenkinsfile to invoke the script after the build by adding the following stage:

```

stage("Upload To MobileIron Core") {
    def host = '<your MobileIron Core host here>'
    def username = '<your MobileIron Core username here>'
    def password = '<your MobileIron Core password here>'
    //only required if the file is not executable
    sh (
        script: "chmod +x ./core_uploadapp.sh"
    )
    sh (
        //sample for upload for an Android app
        script: "./core_uploadapp.sh ${host} ${username} '${password}'
./app/build/outputs/apk/app-debug.apk"

        //it will be like this for uploading an iOS app
        //script: "./core_uploadapp.sh ${host} ${username} '${password}'
./helloworld-ios/build/Debug-iphoneos/myapp.ipa"
    )
}
...

:leveloffset!

```