Red Hat Mobile Application Platform Cheat Sheet

RED HAT DEVELOPER PROGRAM

Table of contents

Red Hat JBoss Enterprise Application Platform 7 (JBoss EAP) is a Java EE 7-certified application platform. It is built on a collection of open source technologies, including an embeddable web server, messaging, clustering and high availability, and caching. It can be a single standalone server or have multiple defined domains, with a master server for centralized management, profiles to define configuration, and hosted servers for scale. Using the command-line tool

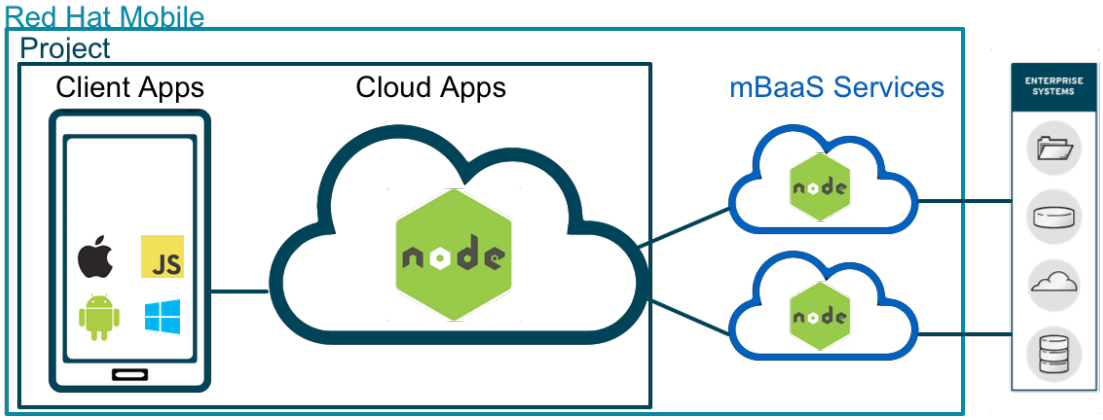
The basics

The Red Hat Mobile Application Platform is a platform for building, deploying and managing mobile applications & their backend integrations. It supports flexible development models, and allows developers to use tooling of their choice. Source code is integrated using Git, and the platform is agnostic to client-side mobile technology stack. The platform categorises its features in the following structure:

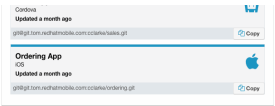
Projects
- Client Apps: Anything which will be deployed on a mobile phone device
- Cloud Apps: Node.js microservices used for all server-side logic specific to this project.

Services & APIs: Re-usable node.js microservices to be used by multiple projects.

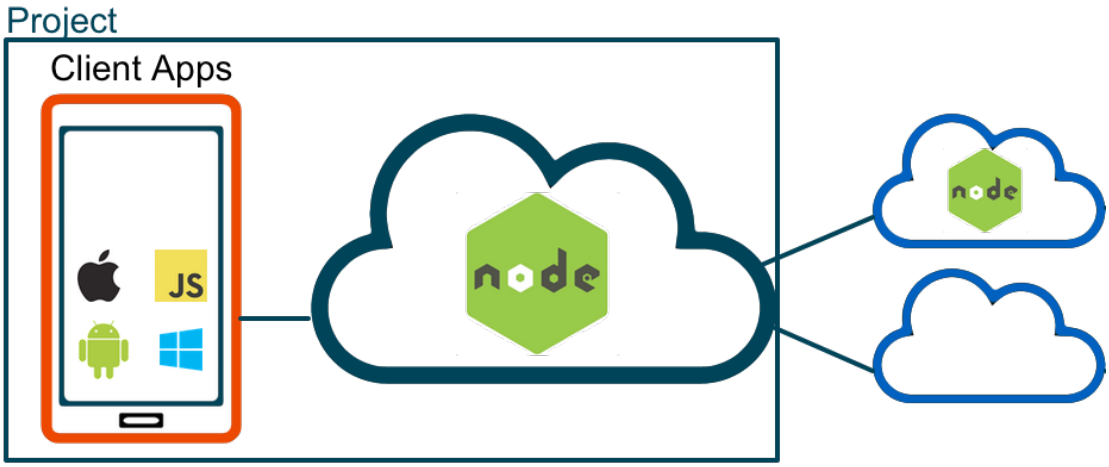
Drag & Drop Apps (D&D herein): Forms-based rapid mobile app development functionality
- Forms: Develop forms with no coding needed. Forms get associated projects.
- Themes: Style D&D forms with an interactive theme builder. Themes get associated with projects.



Comparing client apps, cloud apps, & services

	Client app	Cloud app	mBaaS service
			
Purpose	Building mobile phone apps	Project-specific serverside logic & APIs	Creating re-usable integrations for projects to consume
Programming language	Various (JS, ObjC, Java, Swift, ...)	Node.js	Node.js
Runs on	Mobile phones	Server side	Server side
Project specific?	Yes	Yes	No Reusable across projects
Storage	HTML5 LocalStorage Cordova storage plugins Core Data (iOS) SQLite (Android) ..more	MongoDB Redis	MongoDB Redis Integrate with external databases

Client side SDK & APIs



The client SDK is used to build logic which runs on the mobile phone device. Typically, developers will build user interface manipulation on the client device, and use the $fh.cloud SDK to integrate with the server-side logic they have built.

Client SDKs available

Here's a quick description of the different supported client-side technologies, along with the tooling developers can expect to use with each

	JAVASCRIPT	ANDROID	IOS	.NET
Technology/ Language	Apache Cordova, Appcelerator & other web	Java	Objective C & Swift	.NET, Xamarin
Build farm support	Yes	Yes	Yes	No
Dependency system	NPM	Gradle	CocoaPods	
Auto-init Does the SDK initialize itself automatically?	Yes	No (Call $fh.init manually)	No (Call $fh.init manually)	No (Call $fh.init manually)

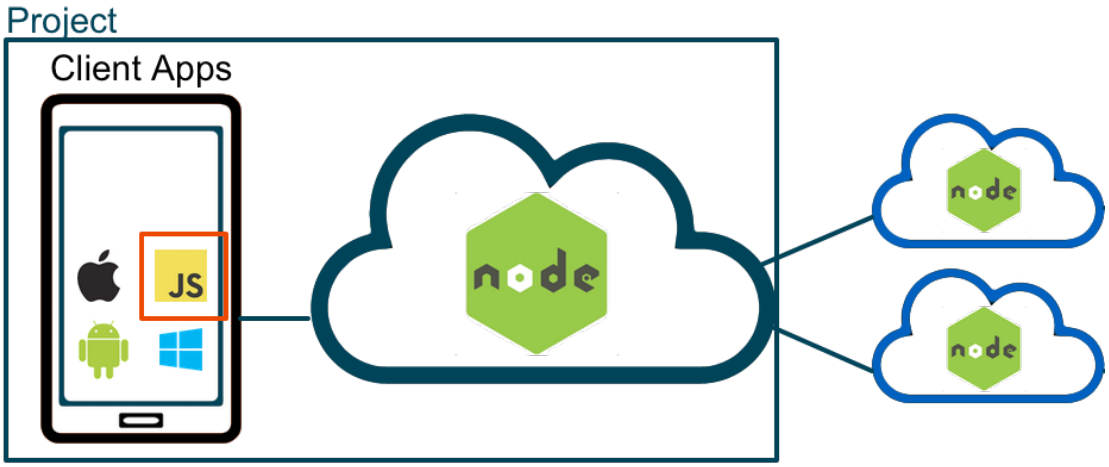
Config file
Each SDK needs this file to tell it how to intialize
	`www`
	`|--fhconfig.json`
		`$PROJECT`
		`|--fhconfig.plist`
			`assets`
			`|--fhconfig.properties`
				`$PROJECT`
				`|--$PROJECT.Shared`
				`|--fhconfig.json`

Available APIs

What APIs are available on what platforms?

$fh.cloud Call a cloud app & execute server-side logic	Yes	Yes	Yes	Yes
$fh.init Initialize the SDK	Yes (but it happens automatically)	Yes	Yes	Yes
$fh.sync	Yes	Yes	Yes	Yes
$fh.push Register & receive device-level push notifications	Yes	Yes	Yes	Yes
$fh.auth	Yes	Yes	Yes	Yes
$fh.sec	Yes	No Native alternatives exist	No Native alternatives exist	No Native alternatives exist
$fh.hash	Yes	No Native alternatives exist	No Native alternatives exist	No Native alternatives exist
$fh.forms	Yes	No	No	No

JavaScript client SDK



Here are some of the most frequently used JavaScript client SDKs. The full list is available, along with examples for other client technologies (Java, Swift, etc) by reading the client API documentation

Function	Code snippet

`$fh.cloud`

```
$fh.cloud({
  path: "/somePath",
  method: "POST|GET|PUT|DELETE", // optional - default GET
  contentType: "application/json", // optional - default
shown
  data: { "username": "testuser"}, // optional - request
body
  timeout: 60000 // optional - default shown
}, function(res) {
  // Success - your response will be in the "res" variable
}, function(msg,err) {
  // Failure
});
```

`$fh.push` Remember to first set one, two, three in fhconfig.json

```
// Registers the app for push notifications.
$fh.push(function(e) {
    // your push message will be in `e.alert`
}, function() {
  // successfully registered
}, function(err) {
  // handle errors
}, { // this next param is optional!
  alias: "user@example.com", // register the device with a
    unique user identifier. Allows pushing to single users
  categories: ["Curling", "Hurling"] // filter only certain
    categories of message to receive
});
```

`$fh.sync`	Initialize the sync service - call this first

```
$fh.sync.init({
  "sync_frequency": 10, // How often to check for new data
in seconds (default shown).
  // .. many more options available
});
```

Then, tell the sync service what dataset(s) to manage. Here, we're registering a dataset called shopping. The Options, Query Params and Metadata can just be empty objects `{}`, but they need to be provided in order.

```
// Manage a dataset called `shopping`
// Options can be provided to override what was provided at
   init time
// Query params can be provided which get passed to the
   server, to filter the dataset retrieved.
// Metadata about this dataset can also be provided, which
   also gets passed to the serverside.
$fh.sync.manage('shopping', {/*options*/}, {/*query
   params*/}, {/* metadata */}, function(){
  // we've registered successfully
});
```

Now that we've registered a dataset, we need to listen to sync events. It's here we'll trigger UI updates to reflect our changes.

```
$fh.sync.notify(function(event) {
  var dataset_id = event.dataset_id; // if you are managing
multiple datasets, it'll be useful to know which this event
relates to
  // The notification message code we are responding to
  switch(event.code){
    case 'sync_complete':
    case 'local_update_applied':
      // at this point, it might make sense to do a list
operation to update the UI using $fh.sync.doList (see
below)
    case 'remote_update_failed':
      // There was an issue updating on the serverside –
handle the error in event.message
    default:
      // there many other notification codes: client_
storage_failed, sync_started, offline_update, collision_
detected, remote_update_applied,
      // local_update_applied, delta_received, record_
delta_received, sync_failed
  }
});
```

Now that we know what events are important to trigger UI update events, let's see what a list operation looks like. This operation does not always trigger a serverside call, so can return very fast with data from the local store. Let's do a list operation on our shopping list dataset.

```
$fh.sync.doList('shopping', function(res) {
  /* where res will be
  {
    '1234' : { // where 1234 is the UID of the record
      hash : '1bd55262212c4ec2ac6ef20d8c8b03b3', // an MD5
               hash of the record
      data : { name : "Oranges" } // the actual record data
    },
    // ... and potentially more records
  }
  */
});
```

We can also create records in our shopping dataset.

```
$fh.sync.doCreate('shopping', { name : "Tomatoes" },
function(res) {
  // create was successful
});
```

`$fh.sync` Initialize the sync service - call this first

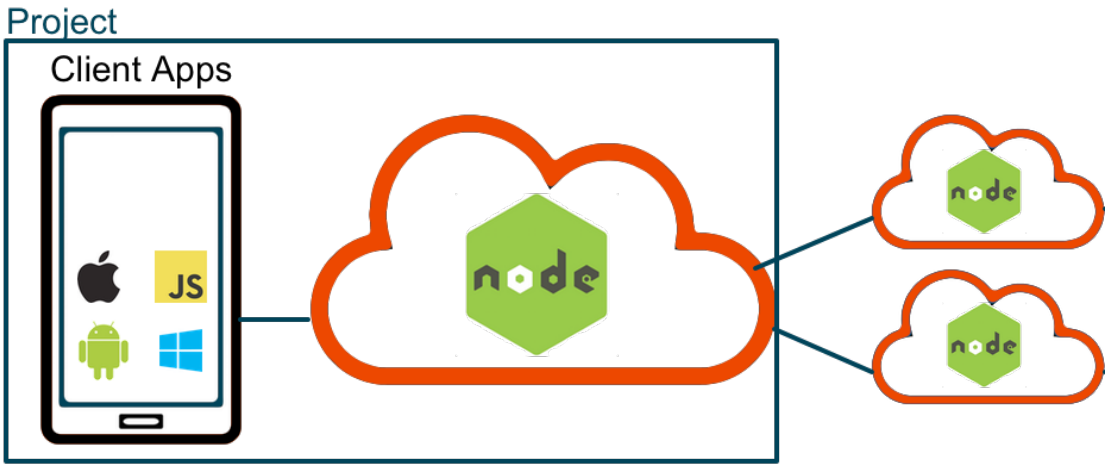
Update, read and delete operations are similar, we just need to provide a record UID after the dataset name:

```
$fh.sync.doUpdate('shopping', 1234, {name : "Grapes"},
 function(){/*success*/});
$fh.sync.doRead('shopping', 1234, function(){/*success*/});
$fh.sync.doDelete('shopping', 1234, function()
{/*success*/});
```

$fh.auth Remember to set the first one, two, three in fhconfig.json

```
// Registers the app for push notifications.
  JavaScript
  // LDAP or Platform User Example
  $fh.auth({
    "policyId": "myLDAP", // UD of an auth policy
       configured in Admin->Auth Policies
    "clientToken": "yourAppId", // get this by doing $fh
       getFHParams().appid
    // When using with Platform or LDAP login:
    "params": { // the parameters associated with the
      requested auth policy
      "userId": "joe@bloggs.com",
      "password": "password"
    }
  }, function (res) {
    var sessionToken = res.sessionToken; // An identifier
      for this session
    var authResponse = res.authResponse; // Auth info
      returned from the auth service - if any
  }, function (msg, err) {
    // something as gone wrong. If err.message ===
"user_purge_data" or "device_purge_data",
    // the user has been flagged for erasing
  });
```

Node.js cloud & mBaas service SDK



As we mentioned above, the cloud application is used for server-side business logic specific to a project.

The mBaaS Services are re-usable components which these Cloud Apps talk to. Both are written using a server-side JavaScript technology called Node.js. The Node.js SDK is published to NPM (Node's dependency management solution) as fh-mbaas-api. You can include it in your cloud apps & mBaaS Services by doing the following:

```
var $fh = require('fh-mbaas-api');
```

Some of the most common Cloud API calls follow, but you can see the full list by visiting our Cloud API documentation.

Function	Code snippet
$fh.service	Call another mBaaS Service. Can be used to call cloudapp-to-service, or service-to-service.

```
$fh.service({
  "guid" : "0123456789abcdef01234567", // The 24 character
unique id of the service
  "path": "/hello", //the path part of the url excluding
   the hostname - this will be added automatically
  "method": "POST",   //all other HTTP methods are
   supported as well. for example, HEAD, DELETE, OPTIONS
  "params": { "hello": "world" }, // request params
  "timeout": 60000, // optional timeout - default shown
  "headers" : {} // optional request headers
}, function(err, body, res) {
  // check err for an error condition - otherwise, the
response is in 'body'
});
```

Cache data in the redis key-value store.

```
var myFavourite = { type : "apples" }; // note JSON values
need to be stringified before savin
  $fh.cache(
    act : "save", key : "favouriteFruit", value : JSON
    stringify(myFavourite), expire : 6
}, function(err, res){
  // check for err - otherwise cache save succeeded.
});
// then read back your value
$fh.cache({
  act : "load", key : "favouriteFruit"
}, function(err, res){
  console.log(JSON.parse(res));
  // res will be { type : "apples " }
});
```

`$fh.db`	Store data in the platform MongoDB. You can also use the MongoDB Node.js Driver directly, but depending on your version of the platform, you may need to "upgrade" your database.

Function	Code snippet
$fh.service	Call another mBaaS Service. Can be used to call cloudapp-to-service, or service-to-service.

```
$fh.db({
  act : "create", type : "fruit", fields : { name :
"apples", price : 10.99 }
}, function(err, data){ /* check for err - otherwise save
succeeded. `data.guid` is the newly created id. */ );
// now read back an item
$fh.db({
  act : "read", type : "fruit", guid :
"4e563ea44fe8e7fc19000002"
}, function(err, data){
  /* Data will be:
  {
    fields : { name : "apples", price : 10.99 },
    guid : "4e563ea44fe8e7fc19000002",
    type : "fruit"
  } */
});
// list all fruit
$fh.db({
  act : "list", type : "fruit"
}, function(err, listResult){ /* listResult.fields contains
an array of data like in the "create" example */ });
// now update the price
$fh.db({
  act : "update", type : "fruit", guid :
"4e563ea44fe8e7fc19000002", fields : { price : 11.99 } //
only the price will change
}, function(err, updateResult){ /* updateResult is like the
"create" example above */ });
// then delete the row
$fh.db({
  act : "delete", type : "fruit", guid :
"4e563ea44fe8e7fc19000002"
}, function(err, deletedEntry){ /* deletedEntry is like the
"create" example above */ });
```

About the author

CIAN CLARKE Cian is a co-founder and principal engineer focused on messaging and virtual assistant technology at ServisBot. An early technologist, he founded his own web consultancy business at 16. Cian was an early member of the original FeedHenry team, where he lead developer evangelism, contributed to engineering and key M&A due diligence prior to exiting to Red Hat. An open source advocate, Cian has authored & contributed to many projects. Prior to this, Cian worked on IBM's social software team, and ran a variety of independent contracts in his consulting shop..

@cianclarke
linkedin.com/in/cianclarke