WHITEPAPER

Building Mobile Apps

Written by Jason Huggins
Director, Enablement & Education at Rocket® Software

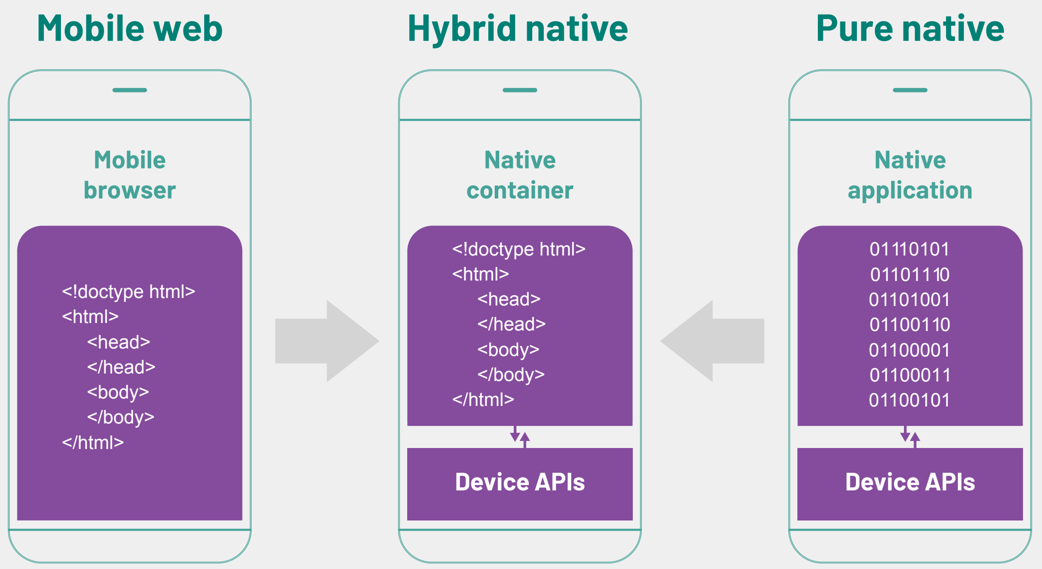
Rocket®
LEGACY POWERS LEGENDARY

Table of contents

Introduction

With Rocket® Uniface, we work to embrace the technological trends that allow the creation of compelling applications. Mobile apps are an important consideration when developing or extending enterprise solutions. This paper looks at some of the options to build a mobile package based on Uniface Dynamic Server Pages.

Mobile development options



Quick to develop, but gives little access to the underlying device features.

Note: A refinement known as Progressive Web Apps uses modern browser enhancements that can access limited device functionality.

A packaged app developed as quicky as Mobile Web, using a single code line, and with access to native device features such as contacts. These do not give a 100% match to the native app user experience, and offline usage requires additional effort.

Peak performance and access to all device features, but these typically have a separate code line per vendor and platform version.

With Rocket Uniface Dynamic Server Pages (DSP) and responsive CSS, you can quickly build responsive screens for mobile web and hybrid apps. For a pure native, Uniface can function as a Mobile Backend as a Service (MBaaS) platform, typically using a RESTful API. This paper will focus on creating and packaging a hybrid solution.

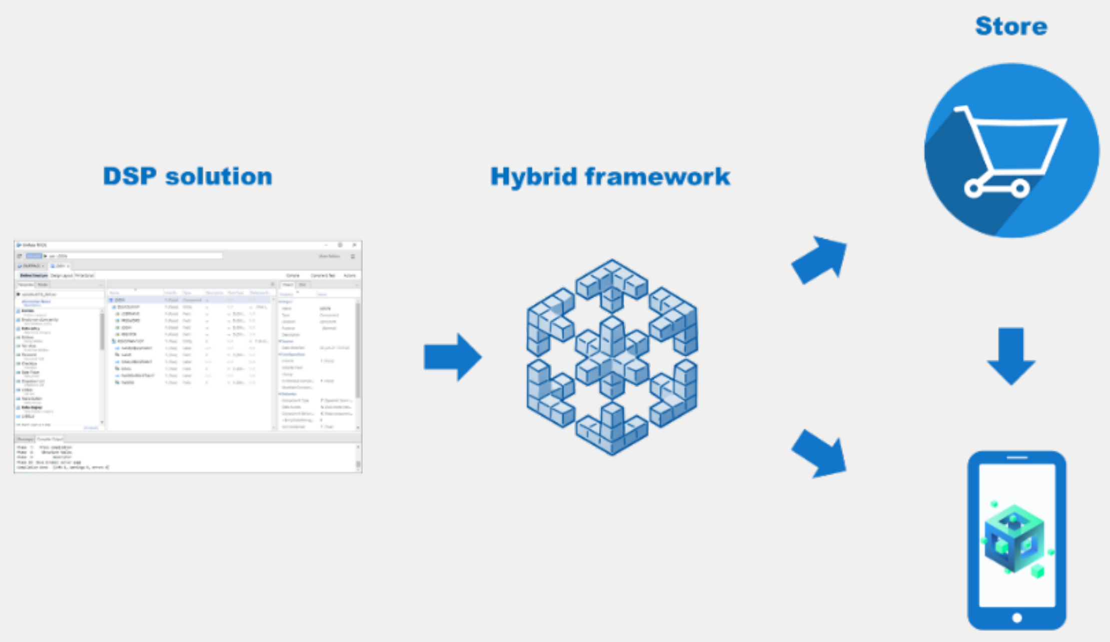
Building hybrid mobile apps

There are many hybrid frameworks that can be used to package and build a Rocket Uniface mobile app. The resultant app can be published to the mobile device directly, which is common for enterprise apps, or published to the relevant stores for public use.

For the purposes of this paper, the Apache Cordova hybrid framework will be used.

See: https://Cordova.apache.org/#getstarted

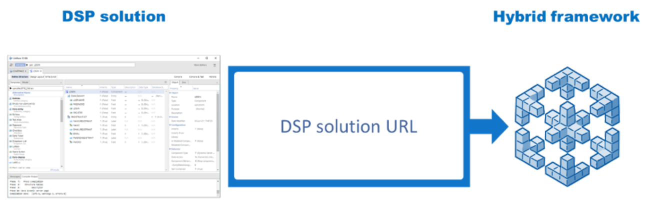
The 'get started fast' page shows a few basic steps that can be adapted for Rocket Uniface apps. This paper will assume that the first step of installing the framework has already been completed. If it hasn't been installed, install now. The example processes that follow will target Google Android as the mobile platform. **Note:** the process is similar for Apple iOS.



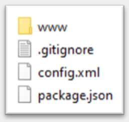
The process step by step

Option 1

In general terms, given a responsive DSP solution, the URL can be specified in the hybrid framework, which is then built ready for deployment. **The steps required to achieve this with Cordova follow:**



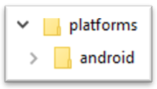
Step 1: To start, use the command line to create a new Cordova project as follows:


www
.gitignore
config.xml
package.json

```
cordova create uniface
```

← Base structure created for the new Cordova project

Step 2: Change to the newly created folder and add the target platform type with:


platforms
android

```
cordova platform add android
```

Step 3: The *www* folder is the web root of the hybrid application. However in this instance the package must point to the DSP web solution. To do this, open ***config.xml*** and change the *src* attribute of the ***content*** element to the URL of the DSP solution.

```
<content src=”https://www.rocketsoftware.com/iot/wrd/MAIN” />
```

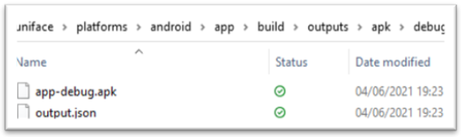
If the server is not yet configured for HTTPS, to allow clear text traffic, the following configuration snippet may be required before the closing *</widget>* tag in *config.xml*.
This setting should not be used for production deployment

```
<edit-config xmlns:android=”http://schemas.android.com/apk/res/android”
  file=”app/src/main/AndroidManifest.xml” mode=”merge” target=”/manifest/application”>
  <application android:usesCleartextTraffic=”true” />
</edit-config>
```

The *id* attribute of the ***widget*** element should be set to a unique value. This is typically done using reverse domain notation e.g. ***com.rocketsoftware.apps.uiot***. Other meta information should also be set by updating the relevant elements such as name, description and author.

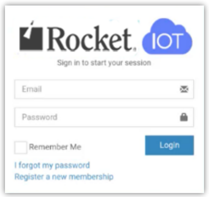
Step 4: The app is now ready for build. Given this example is using the Android platform, an Android emulator is good to have. There are many to choose from, however in this case the default that is part of the Android Studio install will be used. **To build the package and test, simply run the following command:**

```
cordova run android
```


uniface › platforms › android › app › build › outputs › apk › debug

Name	Status	Date modified
app-debug.apk	✓	04/06/2021 19:23
output.json	✓	04/06/2021 19:23

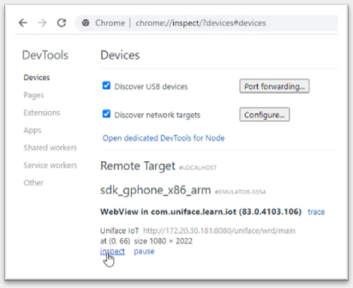
This builds the package, which can be found under the platform folder structure. If installed, the emulator will be launched automatically, the new package deployed, and the app started.



Step 5: Now, to get a view of what is happening on the device and for general debugging, a debugger can be connected. In the case of Android, connecting the Chrome debugger is as easy as visiting the URL:

chrome://inspect/?devices#devices

This will list any devices found that can be debugged, including running emulators. By clicking inspect for the device of interest, a view of the current screen is shown. Alongside this is the ability to debug and inspect the app, in just the same as when working with regular web based application.

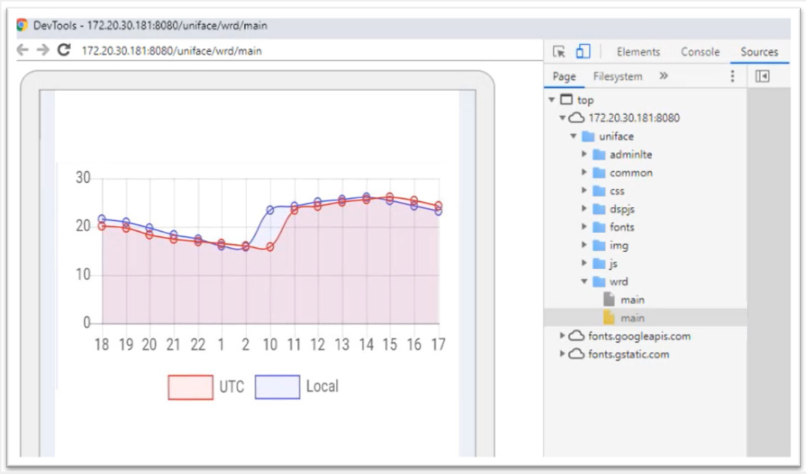

Chrome | chrome://inspect/?devices#devices

DevTools
Devices
Pages
Extensions
Apps
Shared workers
Service workers
Other

Devices
Discover USB devices — Port forwarding...
Discover network targets — Configure...
Open dedicated DevTools for Node

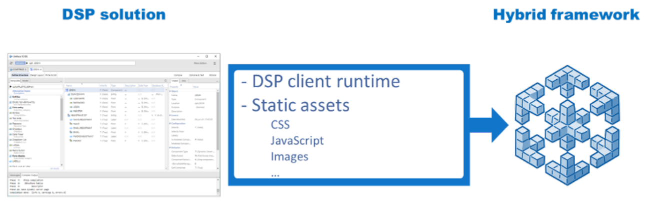
Remote Target #LOCALHOST
sdk_gphone_x86_arm #EMULATOR-5554
WebView in com.uniface.learn.iot (83.0.4103.106) trace
Uniface IoT http://172.20.30.181:8080/uniface/wrd/main
at (0, 66) size 1080 × 2022
inspect pause

The key thing to note in the screen shot below, is that all content is being fetched from the server. There are ways to improve upon this, which will be shown in **Option 2.**

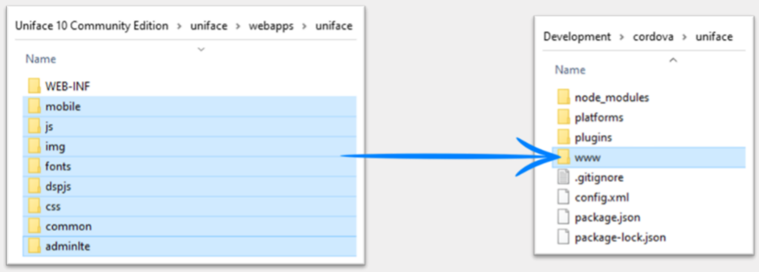

DevTools - 172.20.30.181:8080/uniface/wrd/main
172.20.30.181:8080/uniface/wrd/main

Option 2

An alternative approach to only specifying the server URL in the hybrid framework is to deploy the static content and DSP client runtime. This approach uses the DSP views functionality.



Step 1: To start, copy all of the static assets and the DSP client runtime, from the Uniface web root folder to the *www* folder of the Cordova project created in Option 1. The ***mobile*** folder shown below, has been pre-prepared, containing an ***index.html*** page and JavaScript that loads a DSP view. They are in this subfolder of the web root to ensure relative paths (e.g. *../img/*) resolve to the correct level when run within Cordova.



mobile/index.html contains: *(index page that loads the apps home page using DSP views)*

```
<html>
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="description" content="Load DSP View on Mobile" />
    <meta name="author" content="JH" />
    <meta name="Default" content="Uniface 10">
    <!-- Tell the browser to be responsive to screen width -->
    <meta content="width=device-width, initial-scale=1,
          maximum-scale=1, user-scalable=no" name="viewport">
    <title>Uniface IoT</title>

    <!-- Uniface DSP Runtime-->
    <link rel="stylesheet" type="text/css" href="../css/udsp.css">
    <script type="text/javascript" src="../common/uniface.js" charset="UTF-8"></script>

    <script>
      <!--Use onDeviceReady event to load uniface DSP view-->
      function onDeviceReady() {
        loadUnifaceApp("https://www.rocketsoftware.com/iot/wrd","MAIN","myDiv");
      }
    </script>
    <script type="text/javascript" src="./umob.js" charset="UTF-8"></script>

  </head>
  <body class="hold-transition skin-blue sidebar-mini">

        <div id="myDiv"></div>                  <!-- DIV for app DSP View -->
        <script src="../cordova.js"></script>   <!--Standard Cordova JS-->
  </body>
</html>
```

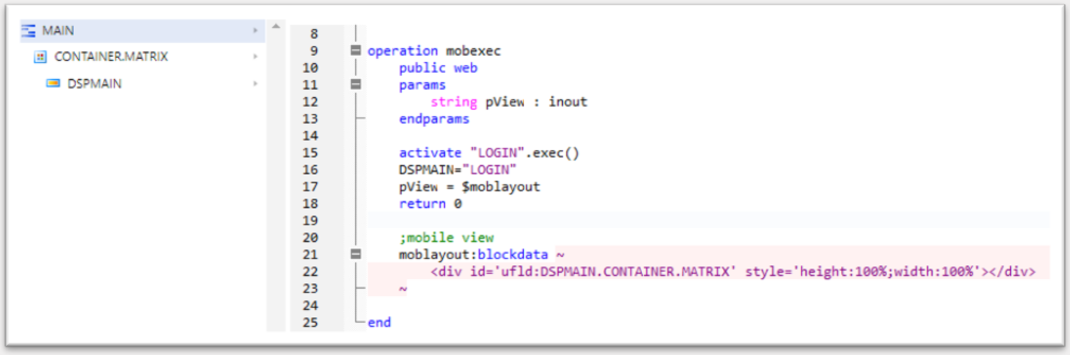
mobile/umob.js contains: *(event listener creation and function to load a DSP view)*

```
//wait for device to be ready before loading DSP View
document.addEventListener('deviceready', onDeviceReady, false);

// load DSP the view
function loadUnifaceApp(pUnifaceWRDURL,pDSP,pDiv) {
  // Define the URL for the Uniface WRD servlet
  window.uniface = window.uniface || {};
  window.uniface.wrdurl = pUnifaceWRDURL;

  // Create instance and show view
  uniface.createInstance(pDSP, pDSP,"mobexec","").then(
    function (r) {
      r.instance.createView(pDiv,r.args[0], document.
getElementById(pDiv)).then(
        function (view) {
            view.show();
        }
      );
    },function (e) {
      console.log(e);
    }
  );
}
```

Step 2: With the static content copied, and the above two files placed in the ***mobile*** subfolder, the parameters of the call to ***loadUnifaceApp*** in ***index.html*** must be set:

- The first parameter is the URL to the WRD e.g. https://{app domain}/{app path}/{wrd servet}
- The second parameter is the start DSP of the solution e.g. MAIN
- The final parameter is the target DIV in the index page so does not need to be changed

Step 3: In ***umob.js***, there is a call to the operation ***mobexec***. This operation must be added to the start DSP to return the initial view. In the example below, the main component has the master container, into which the subsequent DSPs are loaded e.g. the login screen. The view in this case is the bound container HTML element.



```

operation mobexec
    public web
    params
        string pView : inout
    endparams

    activate "LOGIN".exec()
    DSPMAIN="LOGIN"
    pView = $moblayout
    return 0

    ;mobile view
    moblayout:blockdata ~
        <div id='ufld:DSPMAIN.CONTAINER.MATRIX' style='height:100%;width:100%'></div>
    ~

end
```

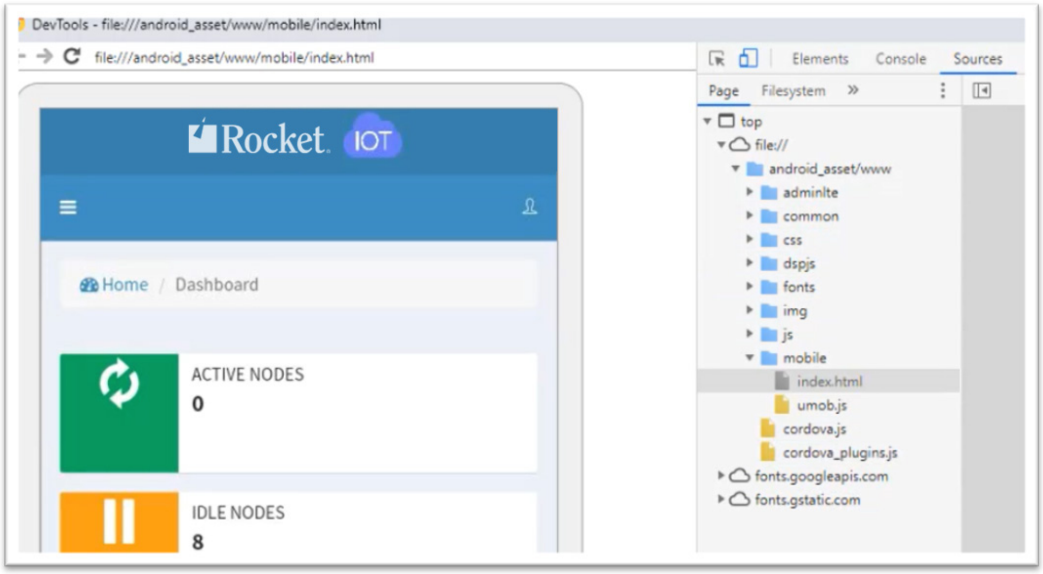
Step 4: Now, to use local web root, open ***config.xml*** and change the ***src*** attribute of the ***content*** element, pointing it at the uniface mobile ***index.html***

```
<content src="mobile/index.html" />
```

Step 5: The app is prepared. To test the changes, rebuild and run with:

```
cordova run android
```

The application runs and behaves as before. It now has a faster startup and better response times given that all static content is already on the device. By connecting the debugger and inspecting the running application, it indeed shows that the static content is all coming from the device.



Now, a question/observation that springs to mind is *"...but doesn't the static content get cached with Option 1 anyway?"* The answer is typically yes; however, **this approach has other advantages:**

- This will generally be quicker for low speed, high latency connections, reducing server load too
- This opens possibilities to explore and utilize various offline techniques.
- Cache management become a lesser issue when updating the application.
- Many hybrid plugins rely on local configuration and content, so this enables all plugins to be used.

Using plugins

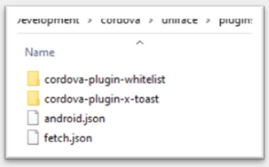
An important characteristic of a hybrid mobile app is the ability to use device functionality and rich plugins. Like everything presented so far, this functionality is also relatively simple. At the time of writing this document, there were over 5000 plugins directly listed on the Cordova site at https://Cordova.apache.org/plugins/. Almost 3000 of these support both Android and iOS. It is fair to say that for the common use cases, a plugin typically exists.

The example will implement a native popup 'toast' message, which is useful for notifications. The specific plugin will be https://www.npmjs.com/package/Cordova-plugin-x-toast. The plugin sites usually provide good instructions and examples.

The process of adding and calling this plugin is as follows:

The first step is to install the plugin using the instructions on the plugin page. In this case there are just two simple commands to run within the Cordova project folder:

```
cordova plugin add cordova-plugin-x-toast
cordova prepare
```



cordova-plugin-whitelist
cordova-plugin-x-toast
android.json
fetch.json

An example call shown on the plugin site is as follows:

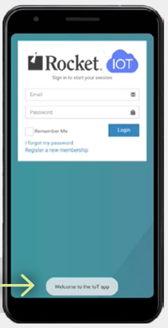
```
window.plugins.toast.showLongBottom(
                  'Hello there!',
                  function(a){console.log('toast success: ' + a)},
                  function(b){alert('toast error: ' + b)});
```

This JavaScript routine can be implemented as a WebOperation in Uniface:

```
weboperation toastMessage               ; display a toast popup
params
        string pMessage : in
endparams
javascript
        window.plugins.toast.showLongBottom(
                pMessage,
                function(a){console.log('toast success: ' + a)},
                function(b){alert('toast error: ' + b)});
endjavascript
end ; toastMessage
```

To test, simply call this WebOperation where required:

```
webactivate $instancename.toastMessage("Welcome to the IoT app")
```



Recap

Packaging a DSP application for mobile deployment, with access to device functionality such as contacts, camera, and geolocation, is a straightforward process. The examples in this paper utilized Cordova and Android, while other platforms follow similar patterns. There are many other hybrid frameworks you may wish to explore.

Deploying all static content from the web server as part of the built package has various advantages. Key benefits are faster startup times and access to a wider range of plugins i.e., those that require local configuration and assets.

The plugins provide access to device functionality. Using Uniface's JavaScript WebOperations, gives an easy way to interface with the plugins. Calling them then becomes just like any other Uniface activation.

To see this in action, please view the recording of the **Rocket Uniface Universe Mobile Webinar on the Uniface YouTube channel.** To learn more about Rocket Uniface development, please visit our free eLearning site.

About Rocket® Uniface

Rocket® Uniface, the most productive, reliable development tool in the industry, provides a model-driven environment for the rapid development of scalable-enterprise mission-critical applications.
Learn more at www.rocketsoftware.com.

ASK AN EXPERT

Rocket®
LEGACY POWERS LEGENDARY