Comp 322/422 - Software Development for Wireless and Mobile Devices

Fall Semester 2017 - Week 10

Dr Nick Hayward

notes app - load initial notes

- start to add some data for the initial notes
- IndexedDB allows us to simply store our objects in their default structure
  - *simply store JavaScript objects directly in our IndexedDB database*
- use transactions when working with data and IndexedDB
- transactions help us create a bridge between our app and the current database
  - *allowing us to add our data to the specified object store*
- use the `readwrite` operation on our previous object store, `ntos`

```
var dbTransaction = openDB.transaction(["ntos"],"readwrite");
```

- use it to retrieve the object store for our data

```
var dataStore = dbTransaction.objectStore("ntos");
```

notes app - load initial notes

- set the schema for the note objects

```
// note
var note = {
  title:title,
  note:note,
  tags:tags
}
// add note
var addRequest = dataStore.add(note,key);
```

- schema matches input fields defined for **create note** form

Cordova app - NoteTaker - v1 - OnsenUI

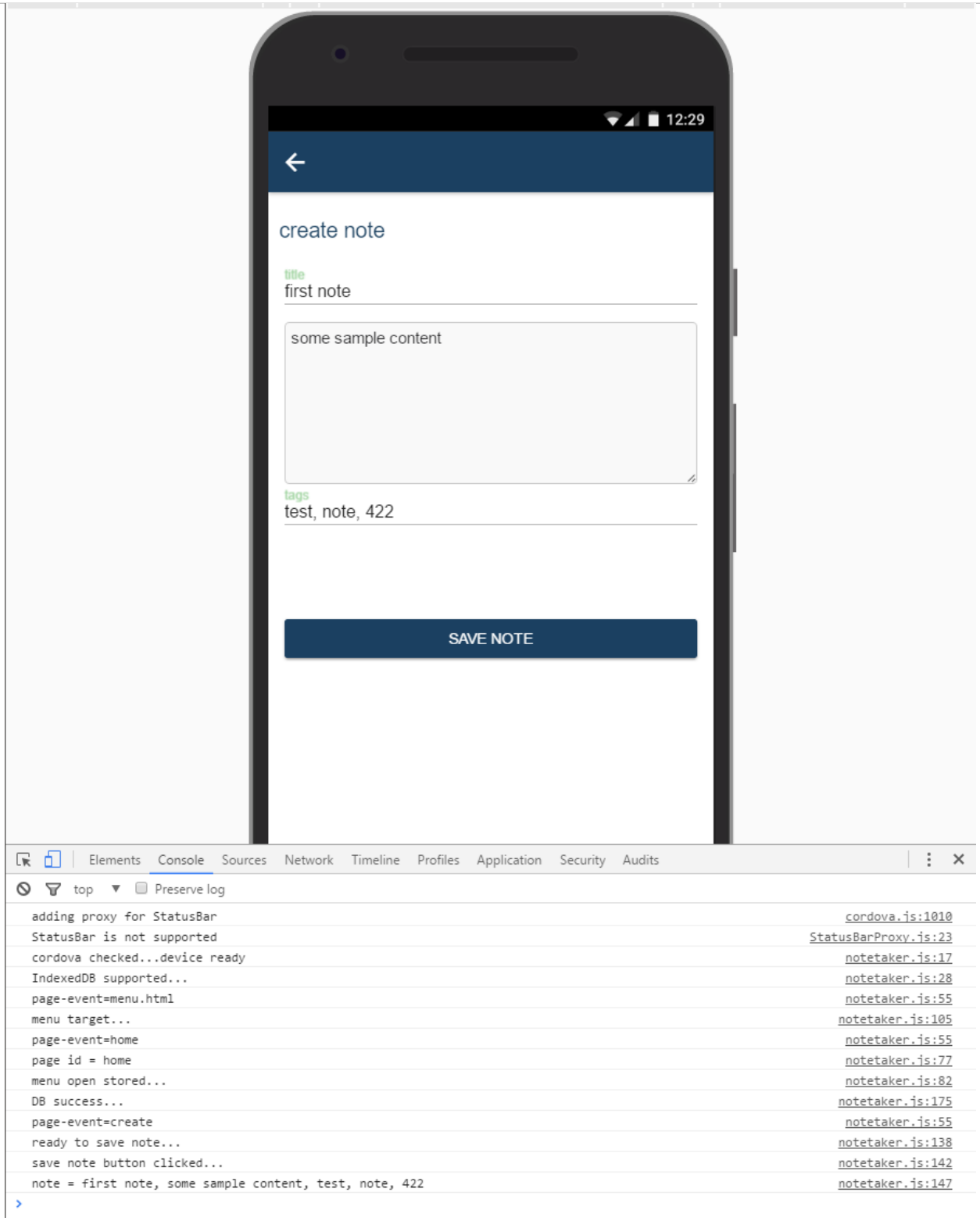
notes app - load initial notes

- add a handler for the **create note** form
- check for the input values
  - *passed to a function to be saved within our database*

```
function createNote(page) {
    //note save handler - check create note page is active...
    if (page.id === "create") {
        console.log("ready to save note...");
        document.getElementById('noteSave').addEventListener('click', function(event) {
            //prevent any bound defaults
            event.preventDefault();
            console.log("save note button clicked...");
            //get values for note - title, content, tags
            var noteTitle = document.getElementById('noteTitle').value;
            var noteContent = document.getElementById('noteContent').value;
            var noteTags = document.getElementById('noteTags').value;
            console.log("note = "+noteTitle+", "+noteContent+", "+noteTags);
      saveNote(noteTitle, noteContent, noteTags);
        });
    }
}
```

- using page events within the app
  - *need to check **create note** page is active, available in the DOM*
  - *before we can start to add listeners for events*
  - *if not, error thrown for the **notesave** element*
- then get values for input fields
- need to validate input before form submission...

Image - check and load IndexedDB



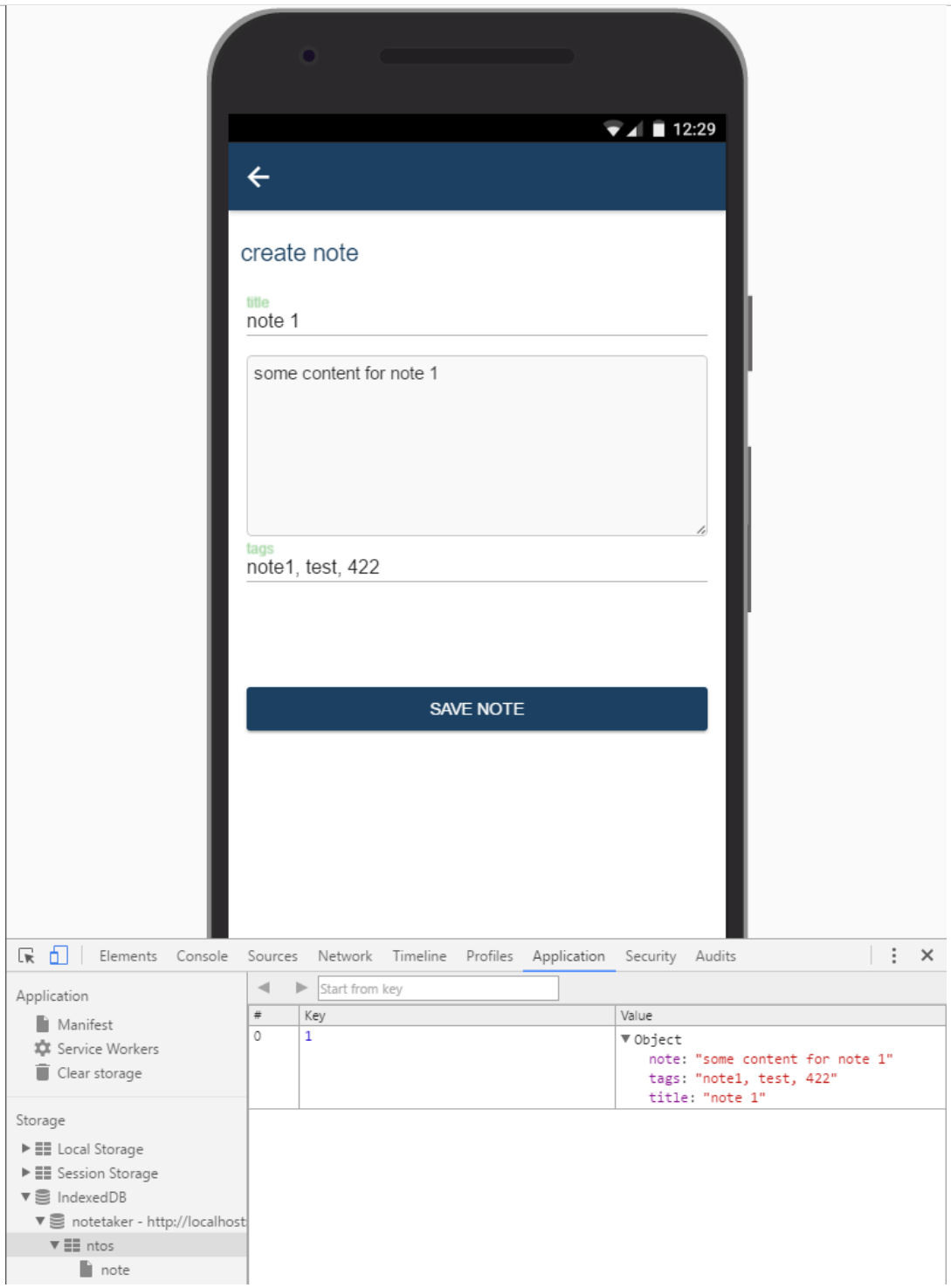
notetaker app - check notesave

notes app - load initial notes

- add our `saveNote()` function
  - *saves note values in specified object store in the database*

```
//save note data to indexeddb
function saveNote(title, content, tags){
    //define a note
    var note = {
        title:title,
      note:content,
        tags:tags
    }
   // create transaction
    var dbTransaction = db.transaction(["ntos"],"readwrite");
    // define data object store
    var dataStore = dbTransaction.objectStore("ntos");
    // add data to store
    var addRequest = dataStore.add(note);
    // success handler
    addRequest.onsuccess = function(e) {
      console.log("data stored...");
      // do something...
    }
    // error handler
    addRequest.onerror = function(e) {
    console.log(e.target.error.name);
      // handle error...
    }
}
```

Image - check and load IndexedDB



notetaker app - save a note

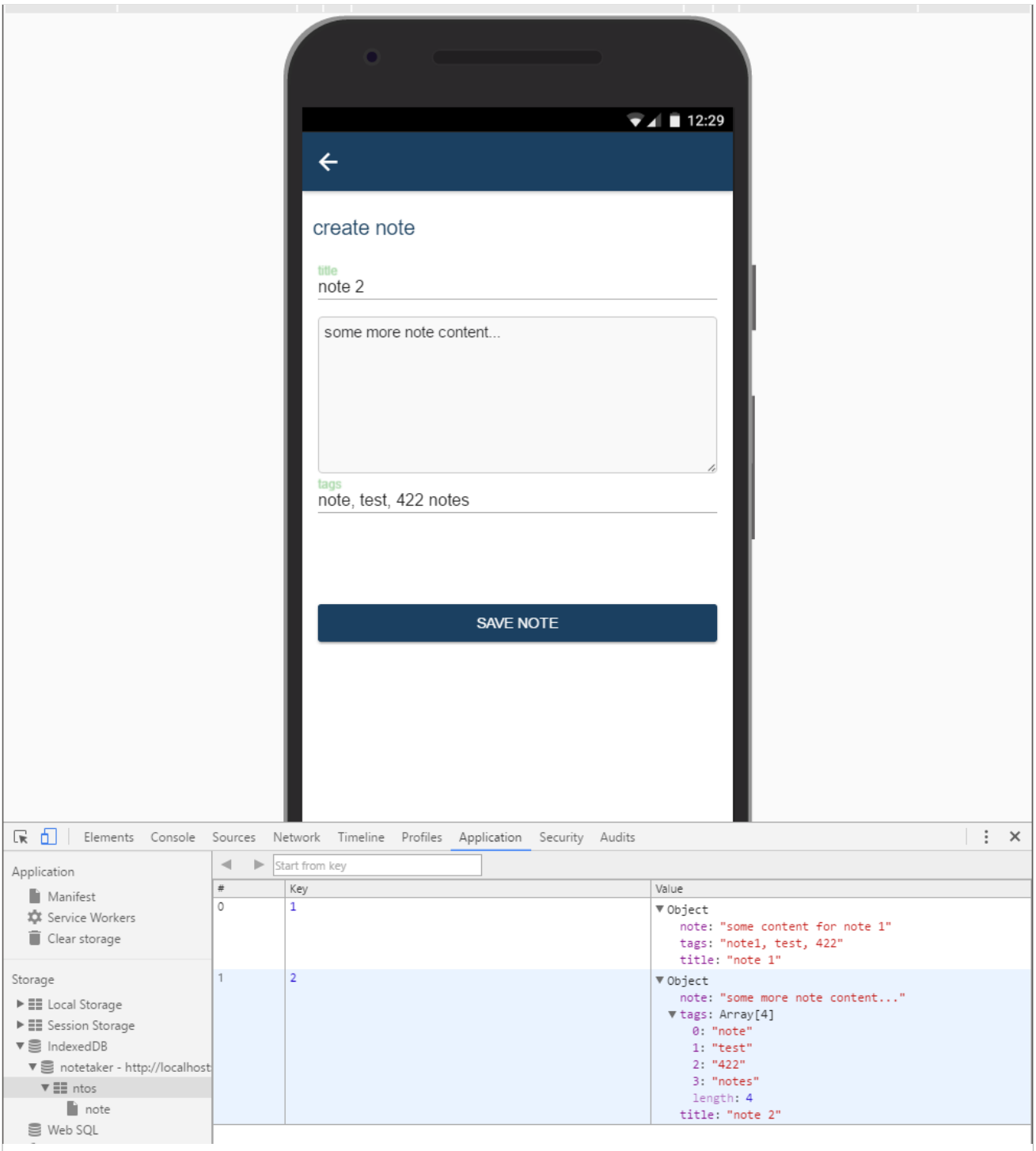
notes app - load initial notes

- need to decide what to do with
  - *tags, success handler, rendering of the new note...*
- as part of the validation for the **tags** input field
  - *we can control structure of input text for tags*
  - *e.g. comma separated, spaces...*
- split each tag from our string
  - *use JS function* `split()`
  - *combine it with a regular expression*

```
...
tags.split(/[ ,]+/);
...
```

- now split our tags from the **create note** form
  - *based on sequence of one or more commas or spaces*

Image - check and load IndexedDB



notetaker app - split tags

notes app - load initial notes

- consider how to handle success event
  - *saving our note data in database's object store*
- might simply return a user to the **create note** form
  - *after showing a notification &c. to provide feedback for saving the note...*
- might return the user to the home page
  - *new note rendered with a feedback message*
- common factors for rendering include the following,
  - *feedback to a user to inform them*
  - *e.g. whether the note was successfully saved or not*
  - *consistent rendering of the notification, buttons, location...*
- consider how to handle error event

Cordova app - NoteTaker - v1 - OnsenUI

notes app - load initial notes

- choose a pattern for **success** event - a saved a note in the database
- show notification with two options
  - ***return to notes*** *and* ***create new note***
- many different patterns available relative to app requirements
- option one - **return to note**
  - *listen for the button's click event*
  - *then dismiss the notification*
  - *pop the* ***create note*** *page from the navigation stack*
  - *return to the home page*
- option two - **create new note**
  - *listen for the button's click event*
  - *dismiss the notification*
  - *reset the form fields*

Cordova app - NoteTaker - v1 - OnsenUI

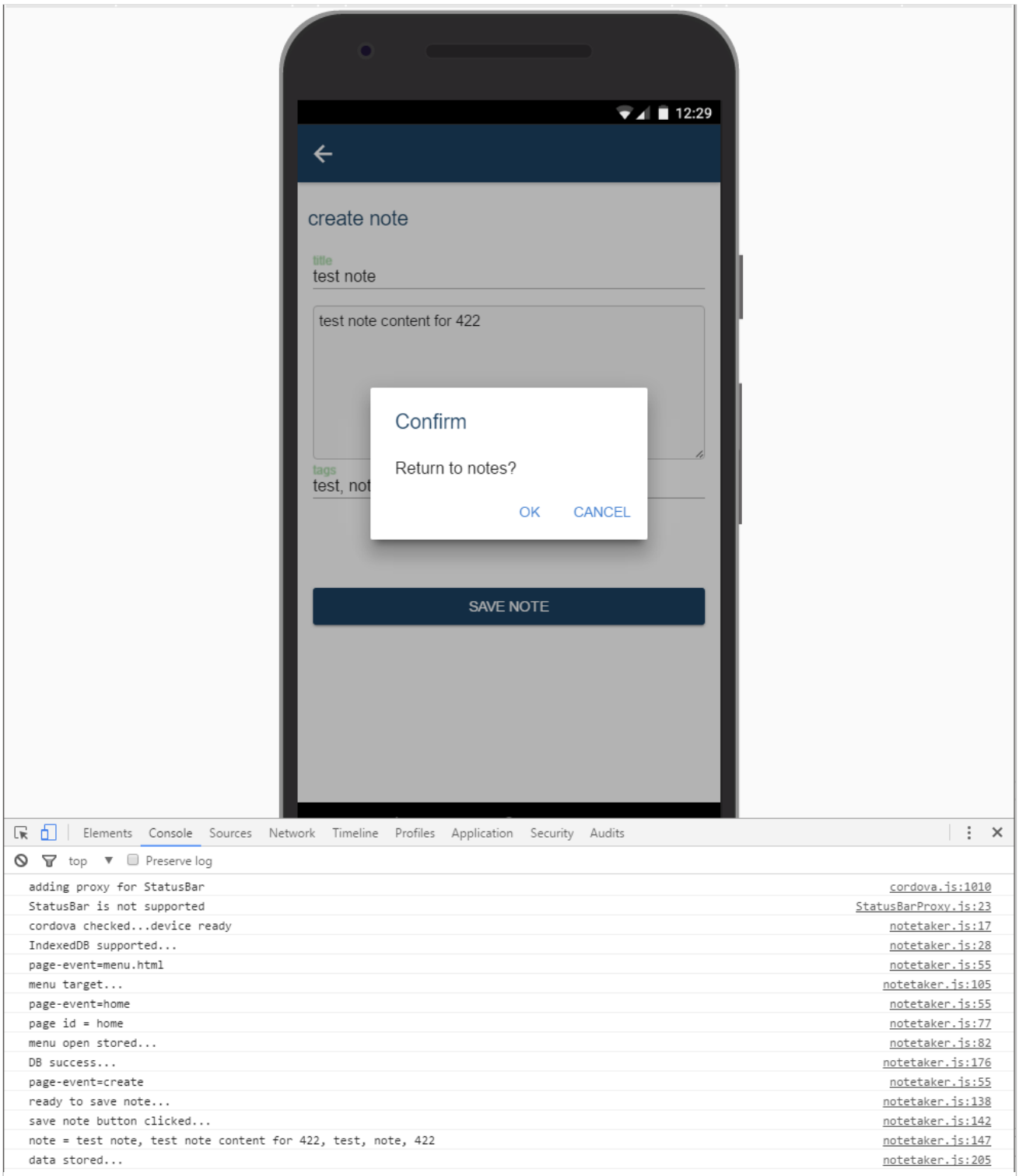
notes app - load initial notes

- add the first option, **return to notes**
  - *as response to user successfully saving a new note*
- need to update the `saveNote()` function

```
...
// success
addRequest.onsuccess = function(e) {
  console.log("data stored...");
  //update user on note stored
  ons.notification.confirm('Return to notes?') // check with user
  .then(function(index) {
    if (index === 1) { // 'ok' button
        document.querySelector('#navigator').popPage(); // return to previous page
    }
  });
}
...
```

- add a notification to Onsen's `ons` object
  - *define it as a confirm notification with message text*
- check user response from button index
- options include
  - *continue with additional new notes*
  - *return to all notes - home page*

Image - save note success



notetaker app - save note success confirmation

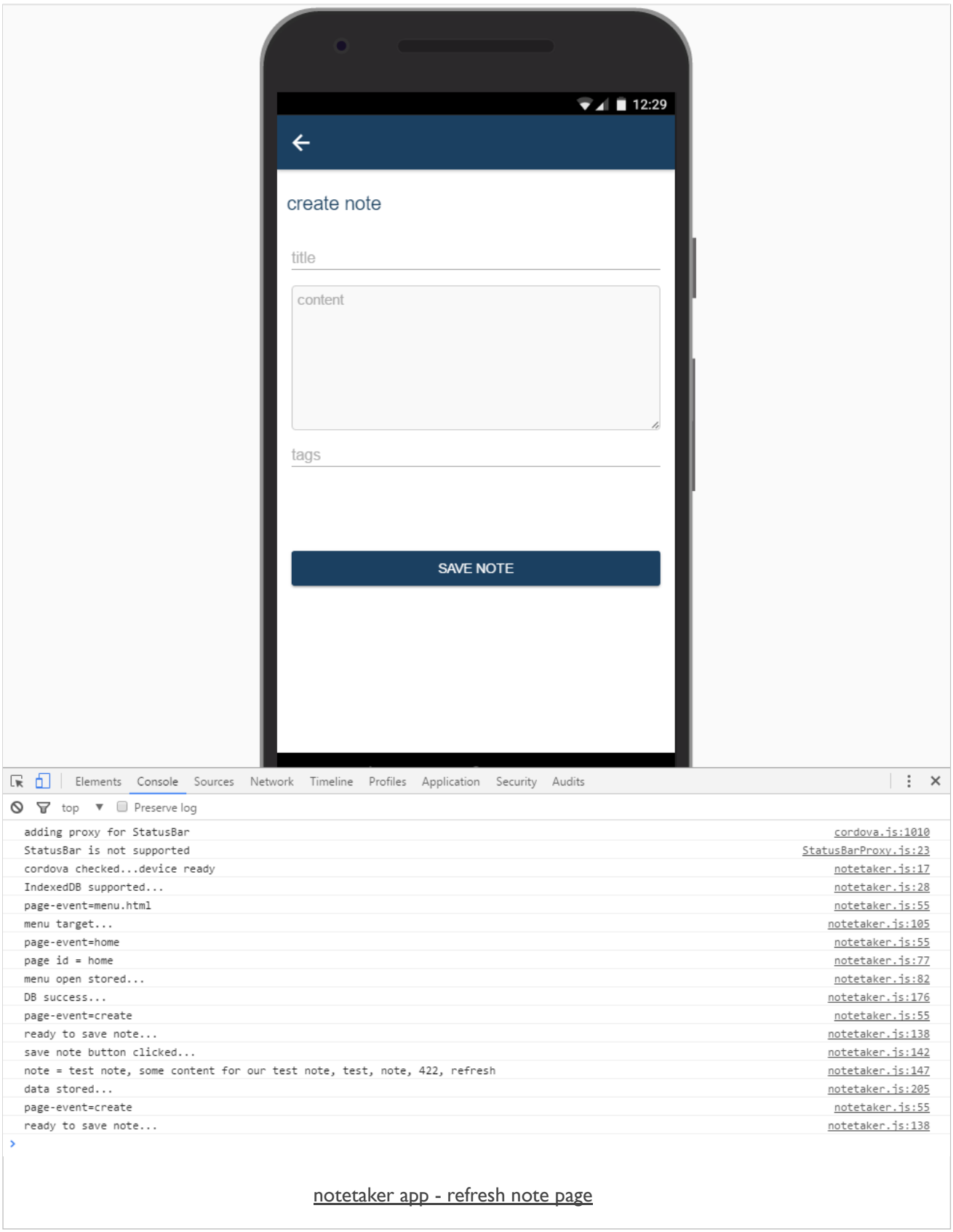
notes app - load initial notes

- need to consider how to handle the current **create note** page
  - *assuming a user selects **cancel** option in the confirmation window*
- user will be returned to the current page in the navigator stack
  - *our **create note** page*
  - *input fields still show previous entry data for note*
- need to ensure that these input fields are cleared
- use existing OnsenUI navigator object to refresh current page

```
//update user on note stored
ons.notification.confirm('Return to notes?') // check with user
.then(function(index) {
    if (index === 1) { // 'ok' button
        document.querySelector('#navigator').popPage(); // return to previous page
  } else if (index === 0) { // check 'cancel' button
        document.querySelector('#navigator').replacePage('create.html', {'animation': 'none'});
    }
});
```

- replace the current top page
  - *set animation for this event to none*

Image - save note success



notetaker app - refresh note page

Extra demos & examples

- extra demos added to course's GitHub account
  - *source - 2017*
  - *source - 2017 - extras*
- cordova examples
  - *maptest*
  - *oauthtest*
  - *splitternav*
  - *sqltest*
- oauth
  - *google test with People API and user's profile*

Extra options - offline ready apps

- a few additional considerations as you prepare your app
  - *for users, testing, publication...*
- a consideration of offline support for our mobile app
- a mobile app needs to consider network usage
  - *with limited or no network connectivity*
  - *poor network reception*
  - *an explicit act by the user to restrict data usage*
- Cordova helps us prepare an app for offline usage
  - *bundles required files as it compiles the app*
- still many considerations for effective offline usage
  - *many disparate parts of our app affected by offline usage*
- not just issue with loss of connectivity to services, data, collaborative features &c
  - *also issue with UI design, interaction, and features*

Extra options - offline ready apps

- changes in state for an app element
- user offline unable to access end service for a request...
- as designers and developers
  - *need to be proactive in removing this option whilst offline*
  - *remove button &c. and option if network connectivity is lost*
  - *update element's state to* `inactive` *& modify interaction*
- act of updating state of an element for offline usage has a number of benefits
- with a disabled state
  - *visual rendering is updated correctly*
  - *event listeners should also become inactive*
  - *we remove any potential issues and errors*
  - *with app logic due to loss of connectivity*
- also offer feedback to the user to inform them
  - *why an element, option, or interaction is no longer available*
  - *inform them of the state of the network...*

Extra options - offline ready apps

- as our Cordova app loads
  - *set a listener for network related events*
- continue to check and monitor status of the network
- trigger changes in state as required during app's lifecycle
  - *app able to respond accordingly simply by checking `online` or `offline` state*
- need to monitor state of the app
  - *user may switch between states of network coverage and usage*
- Cordova provides useful **Network Information** plugin
- plugin has two notable features
  - *1. monitor type of connection our device is currently using*
  - *e.g. `unknown`, `offline`, `wifi`, `4G` ...*
  - *2. respond to events within our app for `offline` and `online`*
- use these events to modify our app and update feedback to users

Extra options - offline ready apps

- need to add the **Network Information** plugin,

```
cordova plugin add cordova-plugin-network-information --save
```

- then check standard `navigator` object for device's connection type
- helps determine user's current connection, e.g. WiFi, 4G, &c
- by monitoring this connection type
  - *we can update our app's UI, interaction, and logic*
- start by adding necessary listeners for network state of our app

```
document.addEventListener("offline", offlineState, false);
document.addEventListener("online", onlineState, false);
```

- these event listeners monitor a change in our app's network status
  - *loss or gain of network connectivity triggers handler*
  - *change in connection type monitored*

Extra options - offline ready apps

- use custom functions
  - *update our app's UI for a disabled or enabled state*
  - *offer feedback to the user...*

```
//handle offline network state
function offlineState() {
  //handle offline network state
    console.log("app is now offline");
    //show ons alert dialog...
    ons.notification.alert('your app is now offline...');
}
```

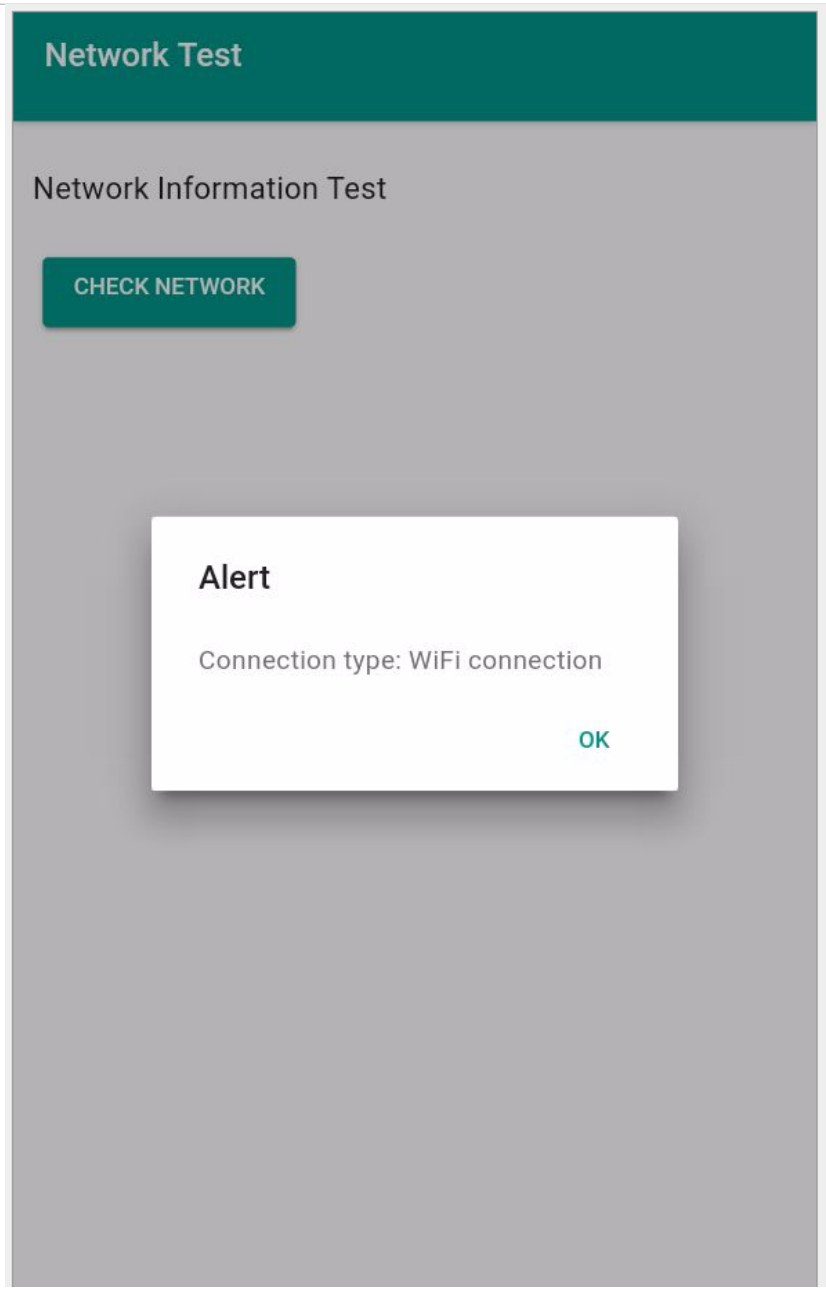
```
//handle online network state
function onlineState() {
// Handle the online event
  var networkState = navigator.connection.type;
    console.log('Connection type: ' + networkState);
  if (networkState !== Connection.NONE) {
    //use connection state to update app, save data &c.
  }
  ons.notification.alert('Connection type: ' + networkState);
}
```

Extra options - offline ready apps

- quickly monitor and check our app's network status and type

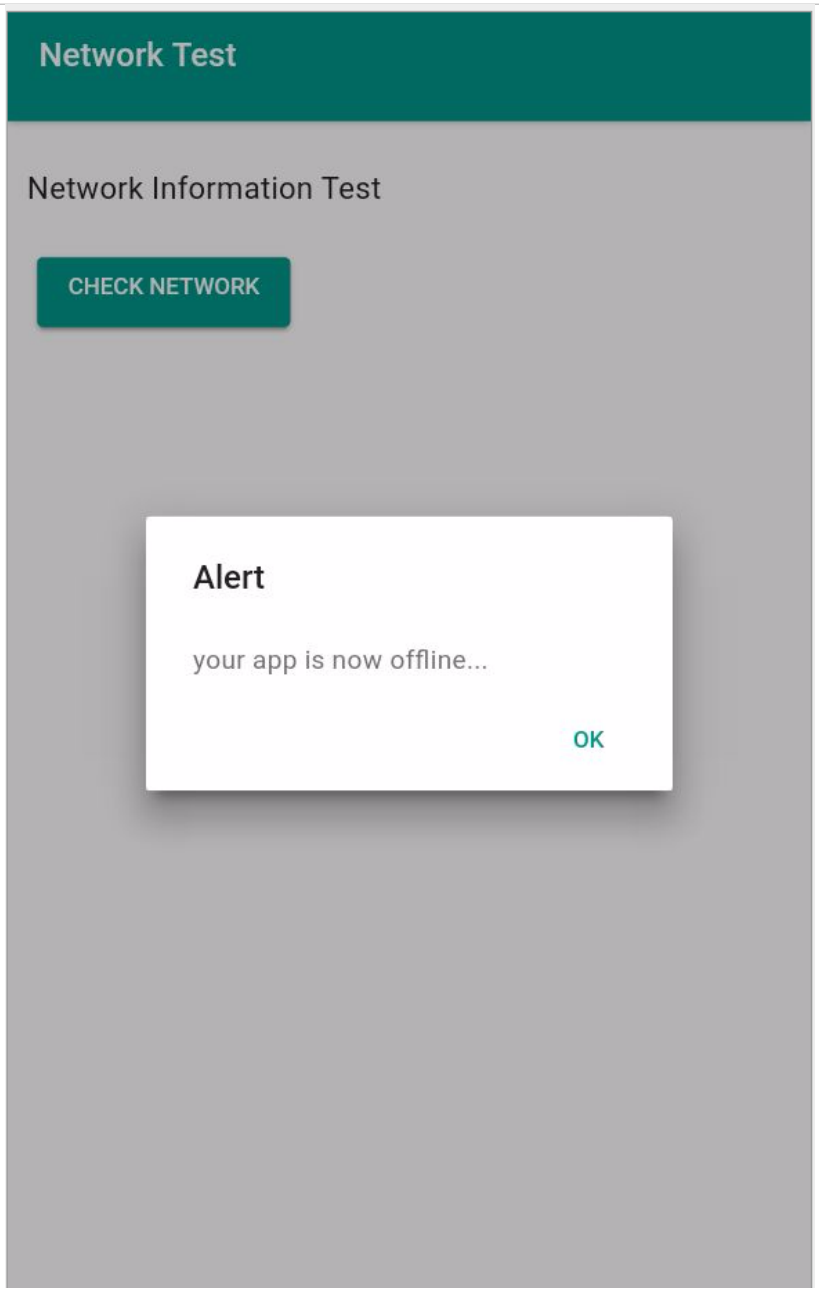
```
function checkConnection() {
  var networkState = navigator.connection.type;
    console.log('check connection requested...');
  var states = {};
  states[Connection.UNKNOWN]  = 'Unknown connection';
  states[Connection.ETHERNET] = 'Ethernet connection';
  states[Connection.WIFI]     = 'WiFi connection';
  states[Connection.CELL_2G]  = 'Cell 2G connection';
  states[Connection.CELL_3G]  = 'Cell 3G connection';
  states[Connection.CELL_4G]  = 'Cell 4G connection';
  states[Connection.CELL]     = 'Cell generic connection';
  states[Connection.NONE]     = 'No network connection';

    console.log('Connection type: ' + states[networkState]);
}
```

Image - Network Information - part 1



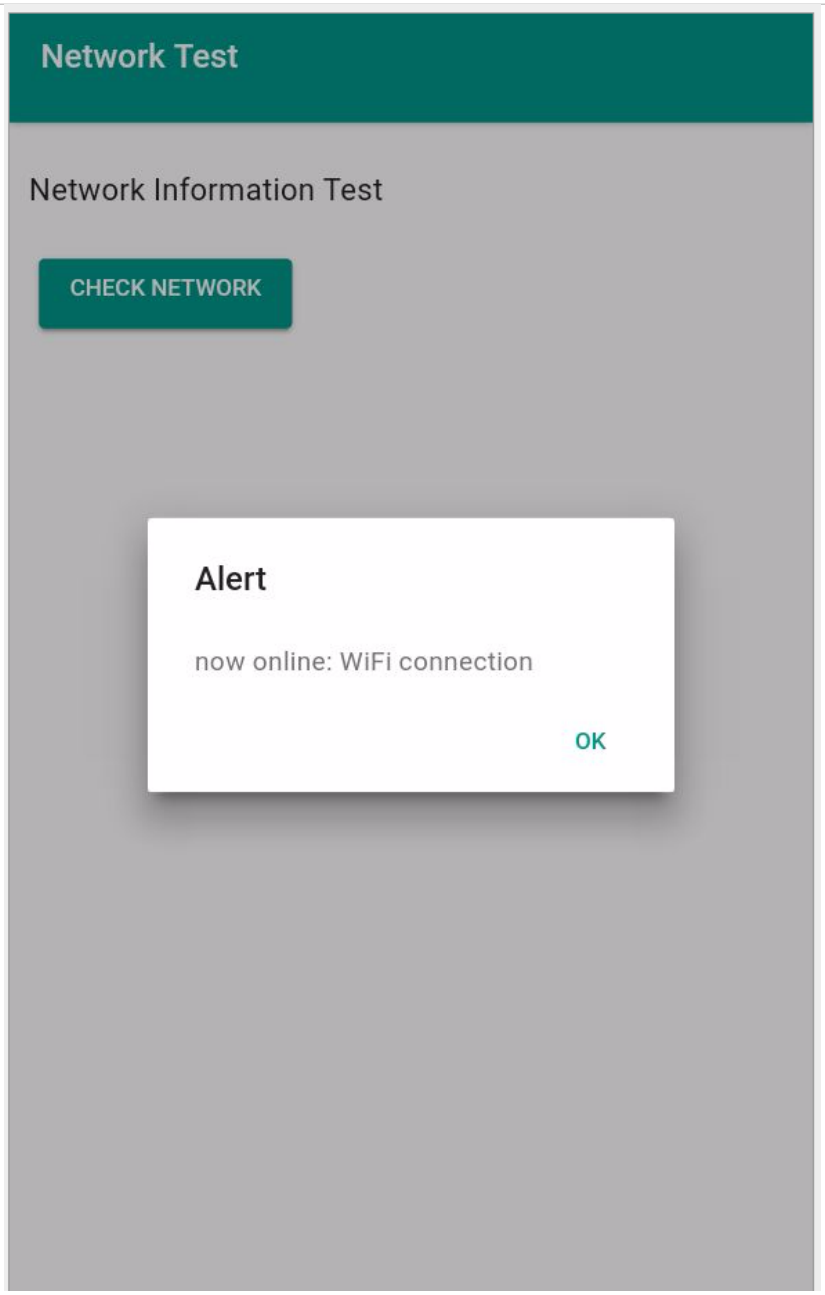
Network Test - connection type

Image - Network Information - part 2



Network Test - check offline

Image - Network Information - part 3



Network Test - check online

Extra options - add International support

- consider publication and release for a mobile app
  - *need to remember international needs and preferences*
  - *different locales, languages, timezones...*
- use Cordova's **globalization** plugin

```
cordova plugin add cordova-plugin-globalization
```

- plugin uses a device's settings
  - *determines user's defined **locale**, **language**, and **timezone***
- e.g. user defined locale of **USA** & language setting of **UK English**
  - *apps will output dates, numbers, measures &c. in a USA compliant format*
  - *& render the language itself using UK English*

Extra options - add International support

- use this plugin with the defined global object
  - *after `deviceready` event*

```
navigator.globalization
```

- start by checking a user's defined language for the current app

```
navigator.globalization.getPreferredLanguage (
  //set success and error callbacks...
  function(language) {
    console.log('language = '+language.value);
  }, function() {
    console.log('error with language check...');
  }
);
```

- check a user's defined locale
  - *same pattern to language check...*

```
navigator.globalization.getLocaleName (
  //set success and error callbacks...
  function(locale) {
    console.log('locale = '+locale.value);
  }, function() {
    console.log('error with locale check...');
  }
);
```

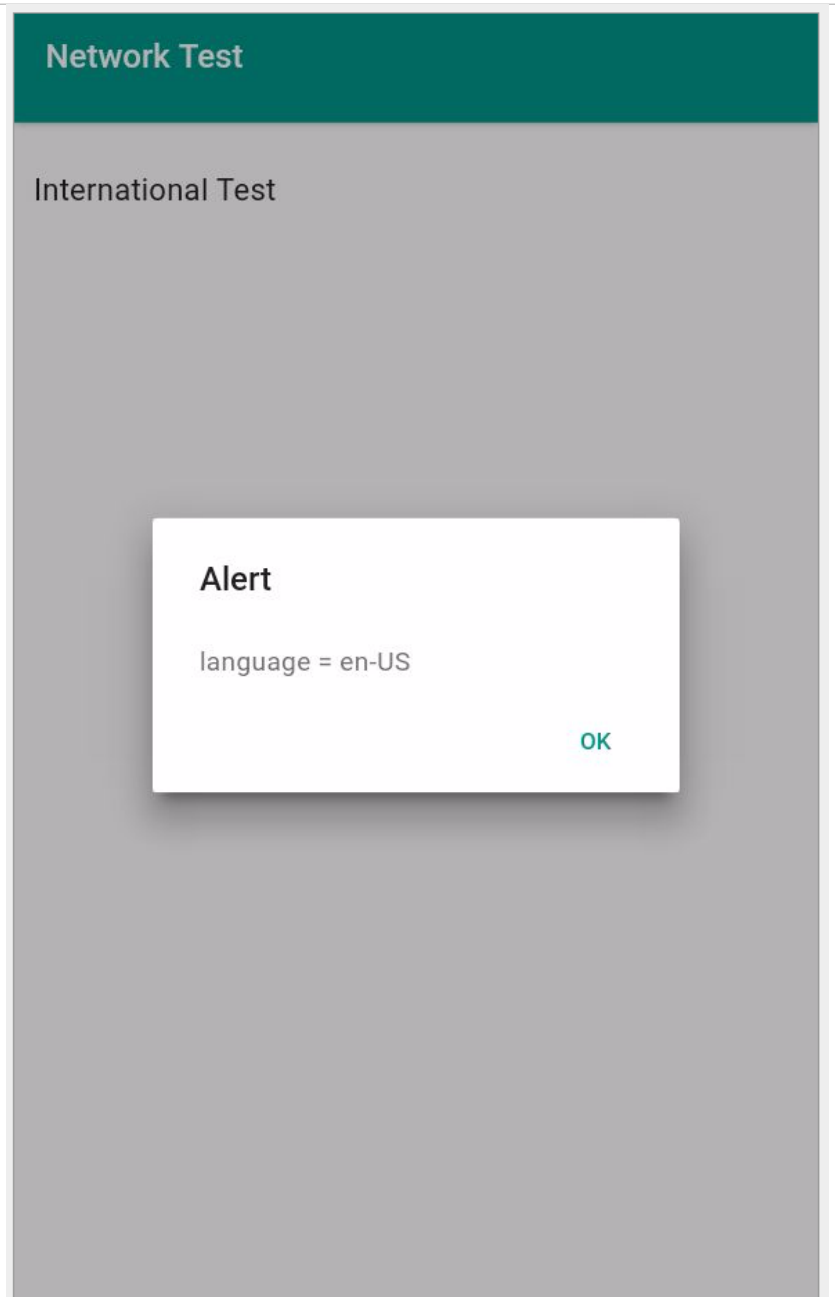
Extra options - add International support

- update and customise our app's dates and times
  - *correctly match the specified locale settings*
- use the `dateToString()` method with the navigator object

```
navigator.globalization.dateToString(
    new Date(),
    function (date) { alert('date: ' + date.value + '\n'); },
    function () { alert('Error getting dateString\n'); },
    { formatLength: 'short', selector: 'date and time' }
);
```

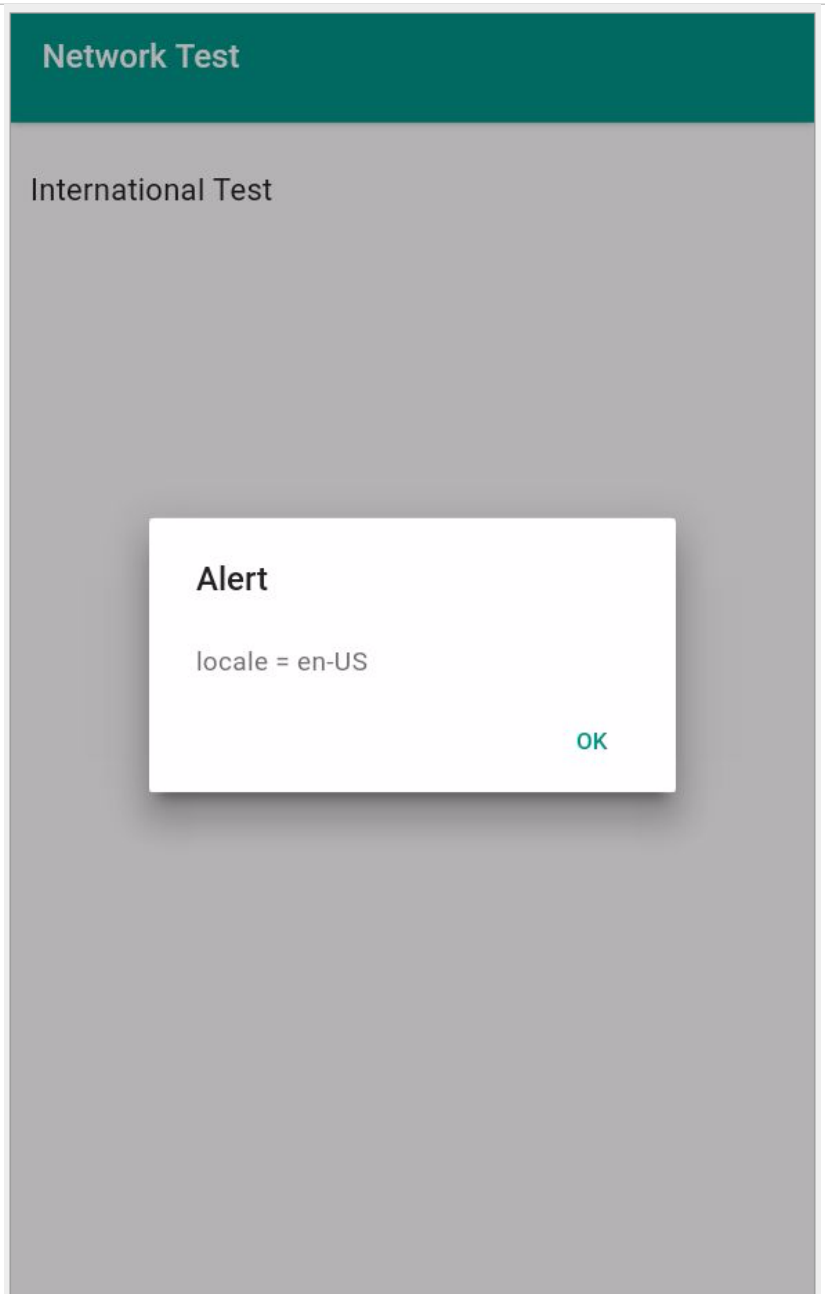
- example from the Cordova API docs - Globalization plugin
- date is created using JavaScript's `Date()` constructor
  - *then use it with `dateToString()` method on navigator object*
  - *ensure rendered date is formatted correctly to match set locale*

Image - International Support - part I



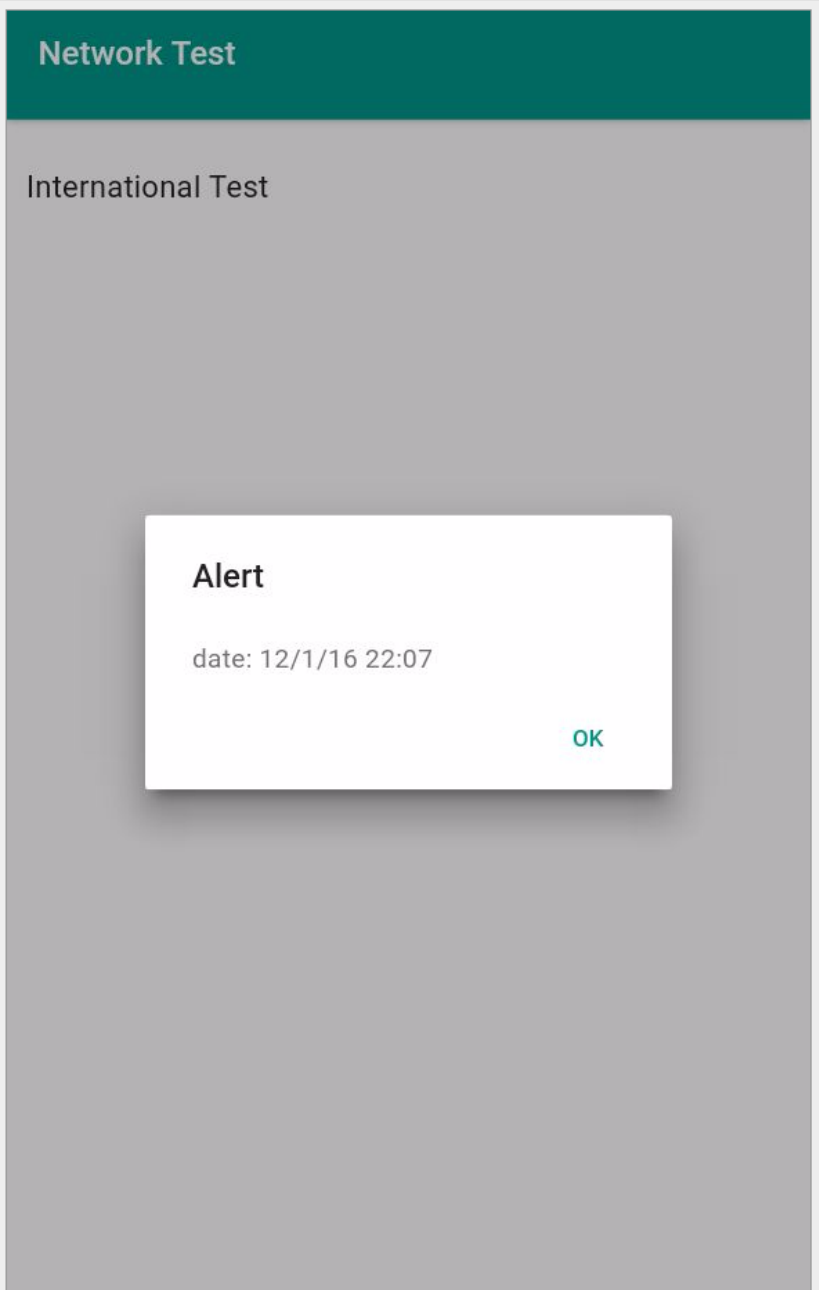
International support - language test

Image - International Support - part 2



International support - locale test

Image - International Support - part 3



International support - date and time test

Extra options - build and customisation - config.xml

- `config.xml` generated as part of Cordova CLI `create` command
- additional preferences we can consider in the metadata
- modify values of these preferences
  - *configure and setup our app with greater precision and customisation*
- Cordova uses `config.xml` file to help setup structures within an app
- standard metadata for author, description, app name, and ID
- additional, useful preferences, e.g.
  - *specifying the default start file as the app loads,*
  - *a security setting for resource access*
  - *a minimum API for building the app*
  - *...*

Extra options - build and customisation - config.xml

- default start file will be specified as `index.html` in the config
- also update this value to a different file,

```
<content src="custom.html" />
```

- also update app's settings to define access privileges and domains for remote resources
  - *e.g. CSS stylesheets, JavaScript files, images, remote APIs, servers...*
  - *specifically remote resources that are not bundled with the app itself*
- Cordova refers to this setting as a **whitelist**
  - *now been moved to a specific plugin*
  - *added by default as we create an app*
- default value for this setting is global access, e.g.

```
<access origin="*" />
```

- this setting will be OK for many apps

Extra options - build and customisation - config.xml

- may need to restrict access, e.g.
  - *due to user input in our app*
  - *remote loading of data*
  - *...*
- might consider restricting our app to specific domains
- add as many `<access>` tags as necessary for our app

```
<access origin="http://www.test.com" />
<access origin="https://www.test.com" />
```

- allows our app to access anything on this domain
  - *including secure and non-secure requests*
- also add subdomains relative to a given domain
  - *simply by prepending a wildcard option*

```
<access origin="http://*.test.com" />
<access origin="https://*.test.com" />
```

- we can now update our app to restrict access to specific, required domains
  - *e.g. remote APIs, servers hosting a DB...*

Extra options - build and customisation - config.xml

- also add further metadata and preferences to help customise our app
- already seen preferences for icons, splashscreens...
- also add further settings for
  - *plugins*
  - *specific installed and supported platforms*
  - *general preferences for all platforms*
  - *or restrict to a single platform*
- for general preferences there are five global options to consider, e.g.
  - *BackgroundColor*
  - *Android and iOS - specific fixed background colour*
  - *DisallowOverscroll*
  - *Android and iOS - prevent a rendered app from moving off the screen*
  - *Fullscreen*
  - *Android (but not iOS) - determine screen usage for an app*
  - *e.g. useful for kiosk style apps...*
  - *HideKeyboardFromAccessoryBar*
  - *iOS (but not Android) - hiding an additional toolbar above a keyboard*
  - *Orientation*
  - *Android (but not iOS) - locking an app's orientation*

Extra options - build and customisation - config.xml

- add any necessary preferences using the `<preference>` element in our `config.xml` file

```
<preference name="fullscreen" value="true" />
```

- add as many preferences as necessary for our app's configuration
- customise our preferences for a specific platform
  - *e.g. restricting a preference to just Android or iOS*

```
<platform name="android">
  <preference name="DisallowOverscroll" value="true" />
</platform>
```

Extra options - build and customisation - merge options

- many Cordova apps developed using a single code base
- with platform specific preferences and UI customisations
- may prefer to create a distinction in the app's design or functionality
- use **merges** options to create platform specific code, files...
- create a new folder called `merges` in our app's root directory
  - *not the www directory*
- use `merges` folder to add platform specific requirements
  - *e.g. css stylesheets*
- add sub-directory to `merges` for each supported platform
- when we build our Cordova app
  - *Cordova will check for a `merges` directory for each platform*
  - *files will replace existing in www directory*
  - *new files added to www directory*

```
config.xml
|-- hooks
|-- merges
    |__ android
    |__ ios
|-- platforms
|-- plugins
|-- www
```

Extra options - build and customisation - merge options

- example usage might include specific stylesheets per platform
- e.g. in our app's `index.html` file add a link to a CSS stylesheet
- stylesheet file added as usual to our app's `www` directory
  - *leave this CSS file blank for the overall project*
- then add matching CSS file to each platform directory in `merges` folder
- CSS file then added to our platform specific app as it is built by Cordova

```
config.xml
|-- hooks
|-- merges
    |__ android
        |__ css
            |__ platform.css
    |__ ios
|-- platforms
|-- plugins
|-- www
    |__ css
        |__ platform.css
    |__ ...
```

- allows us to add specific
  - *styling, layout, and design requirements*
  - *for each supported platform*
- quick and easy option for platform customisation

Extra options - build options - hooks

- we've been using Cordova's CLI tool to help
  - *create our apps, add platforms and plugins, build our apps...*
- we can customise the CLI tool using **hooks**
  - *scripts able to interact with the CLI tool for a given command and action*
- consider **Hooks** in two distinct scenarios
  - *before and after an action is executed by the CLI tool*
- for the CLI tool we might consider adding a **hook**
  - *before or after that command and action is called and executed*
- **hooks** might include automation of standard build options, tools, and commands
- e.g. automation of adding plugins to a project
  - *add a platform, and then add all required plugins using **hook***
- CLI tool checks for **hook** scripts in the `hooks` directory
- to add a **hook**
  - *create a sub-directory in the `hooks` directory - same name as a **hook***
  - *Cordova will then check for scripts to execute*
  - *scripts will be executed in alphabetical order by filename*
- **hooks** can be written in any language supported by the host computer

Extra options - prepare for release

- finalise our Cordova app
- need to consider preparation and packing of the app
  - *ready for publication to one or more app stores*
- each major app store conceptually follows a pattern for release
- to prepare our app for publication
  - *begin by transitioning app from development version to a stable release version*
  - *app requires signing by developer with password*
  - *define ownership of app*
  - *accept responsibility for publication, contents...*
- submit the app to a store for publication
  - *required to provide descriptions for the app itself*
  - *provide a minimum of screenshots for general usage and prominent features*
  - *add supplementary information for publication of app*

Extra options - prepare for release - Play Store

- releasing an Android app is considerably less involved than iOS
  - *developers can release and publish a vast array of application types*
- Play Store - division between preparation of the app, and then publication
- initial preparation
  - *begin by signing our app with a key - create using command line*
  - *use Cordova build tools to create a release build of our app*
- publication to store
  - *upload our app to Google's Play Store for publication*
  - *need to provide some additional supporting information*
  - *title for our app*
  - *icons*
  - *description*
  - *screenshots*
  - *...*
  - *then mark our app as published*

Extra options - prepare for release - signing

- prepare our app for a store
  - *need to sign it using a key store and key prior to publication*
  - *key signs the app, which is saved in the keystore*
- sign our app using the Java tool, **keytool**

```
keytool -genkey -v -keystore my-app-ks.keystore -alias my-app-ks -keyalg RSA -keysize 2048 -validity 10000
```

- command creates both the keystore and key for our app
- command arguments to consider for `-keystore` and `-alias`
- `my-app-ks.keystore`
  - *filename for the keystore*
  - *can be set to a preferred name for your app*
- `my-app-ks`
  - *name of the alias for the keystore*
  - *developer can specify their preferred name*
  - *can be a simple, plain text name for the keystore*

Image - Keytool - Create a Keystore

```
Use "keytool -command_name -help" for usage of command_name
MacBook:networktestprod ancientlives$ keytool -genkey -v -keystore appks.keystore -alias appks -keyalg RSA -keysize 2048 -validity 10000
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Ancient Lives
What is the name of your organizational unit?
  [Unknown]:  Ancientlives
What is the name of your organization?
  [Unknown]:  Ancientlives
What is the name of your City or Locality?
  [Unknown]:  Chicago
What is the name of your State or Province?
  [Unknown]:  Illinois
What is the two-letter country code for this unit?
  [Unknown]:  IL
Is CN=Ancient Lives, OU=Ancientlives, O=Ancientlives, L=Chicago, ST=Illinois, C=IL correct?
  [no]:  yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 10,000 days
        for: CN=Ancient Lives, OU=Ancientlives, O=Ancientlives, L=Chicago, ST=Illinois, C=IL
Enter key password for <appks>
        (RETURN if same as keystore password):
[Storing appks.keystore]
```

Keytools - create a keystore

Cordova app - Plugins

intro

- developing custom plugins for Cordova, and by association your apps
  - *a useful skill to learn and develop*
- it is not always necessary to develop a custom plugin
  - *to produce a successful project or application*
  - *dependent upon the requirements and constraints of the project itself*
- use and development of Cordova plugins is not a recent addition
- with the advent of Cordova 3 plugins have started to change
  - *introduction of Plugman and the Cordova CLI helped this change*
- plugins are now more prevalent in their usage and scope
  - *their overall implementation has become more standardised*

Cordova app - Plugins

structure and design - part 1

- as we start developing our custom plugins
  - *makes sense to understand the structure and design of a plugin*
- what makes a collection of files a plugin for use within our applications
- we can think of a plugin as a set of files
  - *as a group extend or enhance the capabilities of a Cordova application*
- already seen a number of examples of working with plugins
  - *each one installed using the CLI*
  - *its functionality exposed by a JavaScript interface*
- a plugin could interact with the host application without developer input
- majority of plugin designs provide access to the underlying API
  - *provide additional functionality for an application*

Cordova app - Plugins

structure and design - part 2

- a plugin is, therefore, a collection of contiguous files
  - *packaged together to provide additional functionality and options for a given application*
- a plugin includes a `plugin.xml` file
  - *describes the plugin*
  - *informs the CLI of installation directories for the host application*
  - *where to copy and install the plugin's components*
  - *includes option to specify files per installation platform*
- a plugin also needs at least one JavaScript source file
  - *file is used within the plugin*
  - *helps define methods, objects, and properties required by the plugin*
  - *source file is used to help expose the plugins API*

Cordova app - Plugins

structure and design - part 3

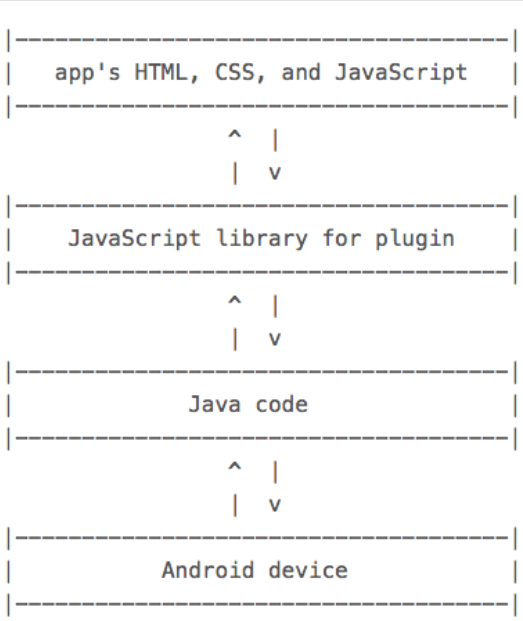
- within our plugin structure
  - *easily contain all of the required JS code in one file*
  - *divide logic and requirements into multiple files...*
- structure depends on plugin complexity and dependencies
- eg: we could bundle other jQuery plugins, handlebars.js. maps functionality...
- beyond the requirement for a `plugin.xml` and plugin JS source file
  - *plugin's structure can be developer specific*
- for most plugins, we will add
  - *native source code files for each supported mobile platform*
  - *may also include additional native libraries*
  - *any required content such as stylesheets, images, media...*

Cordova app - Plugins

architecture - Android

- we can choose to support one or multiple platforms for an application
- consider a plugin for Android
  - *we can follow a useful, set pattern for its development*
- android plugin pattern
  - *application's code makes a call to the specific JS library, API*
  - *plugin's JS then sends a request down the chain*
  - *request sent to specific Java code written for supported versions of Android*
  - *Java code communicates with the native device*
  - *upon success, any return is then handled*
  - *return passed up the plugin chain to the app's code for Cordova*
- bi-directional flow from the Cordova app to the native device, and back again

Image - Cordova Plugin Architecture - Android



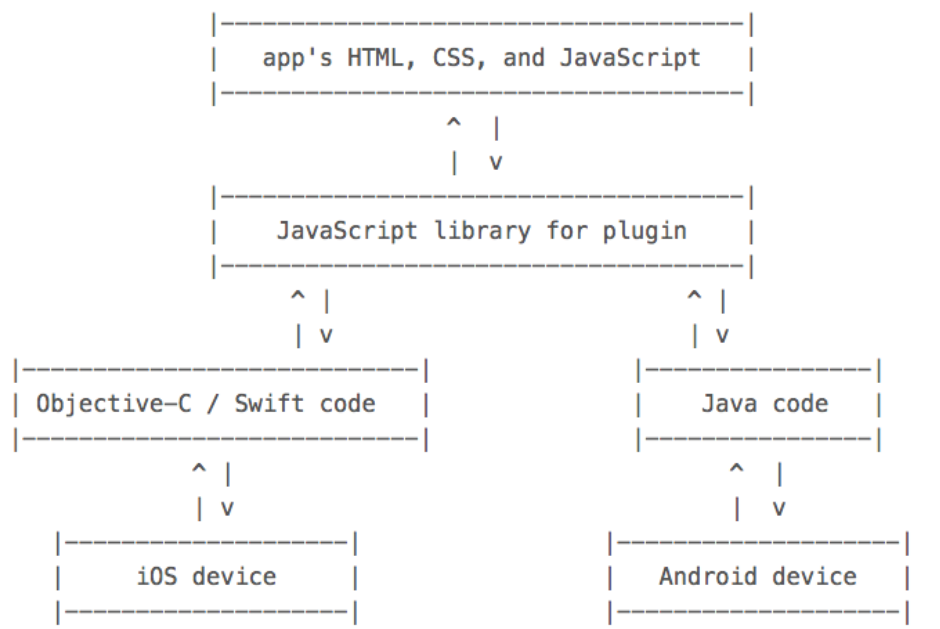
Cordova Plugin Architecture - Android

Cordova app - Plugins

architecture - cross-platform

- update our architecture to support multiple platforms within our plugin design
- maintain the same exposed app content
  - *again using HTML, CSS, and JavaScript*
- maintain the same JavaScript library, API for our plugin
- add some platform specific code and logic for iOS devices
  - *add native Objective-C/Swift code and logic*
- inherent benefit of this type of plugin architecture
  - *the plugin's JavaScript library*
- as we support further platforms
  - *plugin's JavaScript library should not need to change per platform*

Image - Cordova Plugin Architecture - Cross-platform



Cordova Plugin Architecture - Cross-platform

Cordova app - Plugins

Plugman utility - part 1

- for many plugin tasks in Cordova we can simply use the CLI tool
- we can also use the recent *Plugman* tool
  - *useful for the platform-centric workflow*
- *Plugman* tool helps us develop custom plugins
  - *helps create simple, initial template for building plugins*
  - *add or remove a platform from a custom plugin*
  - *add users to the Cordova plugin registry*
  - *publish our custom plugin to the Cordova plugin registry*
  - *likewise, unpublish our custom plugin from the Cordova plugin registry*
  - *search for plugins in the Cordova plugin registry*

Cordova app - Plugins

Plugman utility - part 2

- need to install *Plugman* for use with Cordova
  - *use NPM to install this tool*

```
npm install -g plugman
```

- OS X may need `sudo` to install
- `cd` to working directory for our new custom plugin
  - *now create the initial template*

```
plugman create --name cordova-plugin-test --plugin_id org.csteach.plugin.Test --plugin_version 0.0.1
```

- with this command, we are setting the following parameters for our plugin
  - `--name` = *the name of our new plugin*
  - `--plugin_id` = *sets an ID for the plugin*
  - `--plugin_version` = *sets the version number for the plugin*
- also add optional metadata, such as author or description, and path to the plugin...
- new plugin directory containing
  - `plugin.xml`, *www directory,* `src` *directory*

Cordova app - Plugins

Plugman utility - part 3

- using `plugman`, we can also add any supported platforms to our custom plugin

```
// add android
plugman platform add --platform_name android
// add ios
plugman platform add --platform_name ios
```

- command needs to run from the working directory for the custom plugin
- template creates plugin directories

```
|- plugin.xml
|- src
   |- android
      |- Test.java
|- www
   |- test.js
```

- three important files that will help us develop our custom plugin
  - *`plugin.xml` file for general definition, settings...*
  - *`Test.java` contains the initial Android code for the plugin*
  - *`test.js` contains the plugin's initial JS API*

Cordova app - Plugins

Plugman utility - part 4

- now update plugin's definition, settings in `plugin.xml` file
  - *helps us define the general structure of our plugin*
- within the `<plugin>` element, we can identify our plugin's metadata
  - *`<name>`, `<description>`, `<licence>`, and `<keywords`*
- need to clearly define and structure our JS module
  - *corresponds to a JS file for our plugin*
  - *helps expose the plugin's underlying JS API*
- `<clobbers>` element is a sub-element of `<js-module>`
  - *inserts JS object for plugin's JS API into application's window*
- update `target` attribute for `<clobbers>` adding required window value

```
<clobbers target="window.test" />
```

- now corresponds to object defined in `www/test.js` file
- exported into app's window object as `window.test`
  - *access underlying plugin API using this `window.test` object*

Cordova app - Plugins

Test plugin 1 - JS plugin - part 1

- majority of Cordova plugins include native code
  - *for platforms such as Android, iOS, Windows Phone...*
  - *not a formal requirement for plugins*
- start by developing our custom plugin using JavaScript
  - *eg: create a custom plugin to package a JavaScript library*
  - *or a combination of libraries*
  - *create a structured JS plugin for our application*
- start by creating a simple JavaScript only plugin
  - *helps demonstrate plugin development*
  - *general preparation and usage*
- need to quickly update our `plugin.xml` file
  - *correctly describe our new plugin*

```
<description>output a daily random travel note</description>
```

Test plugin 1 - JS plugin - part 2

- now start to modify our plugin's main JS file, `www/test.js`
- use this JS file to help describe the plugin's primary JS interface
  - *developer can call within their Cordova application*
  - *helps them leverage the options for the installed plugin*
- by default, when Plugman creates a template for our custom plugin
  - *includes the following JS code for `test.js` file*

```
var exec = require('cordova/exec');

exports.coolMethod = function(arg0, success, error) {
    exec(success, error, "test", "coolMethod", [arg0]);
};
```

Cordova app - Plugins

Test plugin 1 - JS plugin - part 3

- part of the default JS code
  - *created based upon the assumption we are creating a native plugin*
  - *eg: for Android, iOS platforms...*
- loads the `exec` library
  - *then defines an export for a JS method called `coolMethod`*
- as we develop a native code based plugin for Cordova
  - *need to provide this method for each target platform*
- working with a JS-only plugin, simply export a function for our own plugin
- now update this JS file for our custom plugin

```
module.exports.dailyNote = function() {
return "a daily travel note to inspire a holiday...";
}
```

- to be able to use this plugin
  - *a Cordova application simply calls `test.dailyNote()`*
  - *the note string will be returned*

Test plugin 1 - JS plugin - part 4

- simply exposing one test method through the available custom plugin
- easily build this out
  - *expose more by simply adding extra exports to the `test.js` file*
- also add further JS files to the project
  - *also export functions for plugin functionality*
- need to update our plugin to work in an asynchronous manner
  - *a more Cordova like request pattern for a plugin*
- when the API is called
  - *at least one callback function needs to be passed*
  - *then the function can be executed*
  - *then passed the resulting value*

Cordova app - Plugins

Test plugin 1 - JS plugin - part 5

```
module.exports = {

  // get daily note
  dailyNote: function() {
      return "a daily travel note to inspire a holiday...";
  },

  // get daily note via the callback function
  dailyNoteCall: function (noteCall) {
    noteCall("a daily travel note to inspire a holiday...");
  }
};
```

- exposing a couple of options for requests to the plugin
- now call `dailyNote()`
  - *get the return result immediately*
- call `dailyNoteCall()`
  - *get the result passed to the callback function*

Test plugin 1 - JS plugin - part 6

- now need to test this plugin, and make sure that it actually works as planned
- first thing we need to do is create a simple test application
  - *follow the usual pattern for creating our app using the CLI*
  - *add our default template files*
  - *then start to add and test the plugin files*

```
cordova create customplugintest1 com.example.customplugintest1 customplugintest1
```

- also add our required platforms,

```
cordova platform add android
```

Cordova app - Plugins

Test plugin 1 - JS plugin - part 7

- we can then add our new custom plugin

```
cordova plugin add ../custom-plugins/cordova-plugin-test
```

- currently installing this plugin from a relative local directory
- when we publish a plugin to the Cordova plugin registry
  - *install custom plugin using the familiar pattern for standard plugins*
- we can now check the installed plugins for our custom plugin

```
cordova plugins
```

Image - Cordova Custom Plugin

```
Drs-MacBook-Air-2:customplugintest1 ancientlives$ cordova plugins
cordova-plugin-whitelist 1.0.0 "Whitelist"
org.csteach.plugin.Test 1.0.0 "Test"
Drs-MacBook-Air-2:customplugintest1 ancientlives$
```

Cordova Installed Plugins

Cordova app - Plugins

Test plugin 1 - JS plugin - part 8

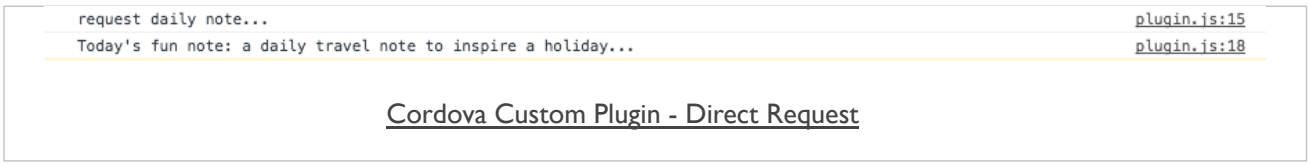
- now need to setup our home page,
- add some jQuery to handle events
- then call the exposed functions from our plugin
- start by adding some buttons to the home page

```
<button id="dayNote">Daily Note</button>
<button id="dayNoteSync">Daily Note Async</button>
```

- then update our app's `plugin.js` file
  - *include the logic for responding to button events*
  - *then call plugin's exposed functions relative to requested button*

```
//handle button tap for daily note - direct
$("#dayNote").on("tap", function(e) {
  e.preventDefault();
  console.log("request daily note...");
  var note = test.dailyNote();
  var noteOutput = "Today's fun note: "+note;
  console.log(noteOutput);
});
```

Image - Cordova Custom Plugin



Cordova app - Plugins

Test plugin 1 - JS plugin - part 9

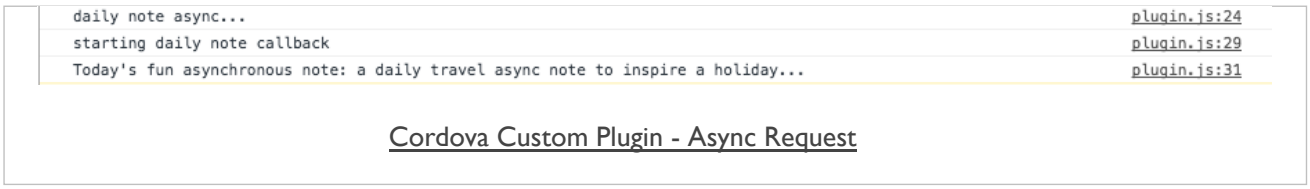
- request asynchronous version of daily note function from plugin's exposed API
- add an event handler to our `plugin.js` file
  - *responds to the request for this type of daily note*

```
//handle button press for daily note - async
$("#dayNoteSync").on("tap", function(e) {
  e.preventDefault();
  console.log("daily note async...");
  var noteSync = test.dailyNoteCall(noteCallback);
});
```

- then add the callback function

```
function noteCallback(res) {
  console.log("starting daily note callback");
  var noteOutput = "Today's fun asynchronous note: "+ res;
  console.log(noteOutput);
}
```

Image - Cordova Custom Plugin



Cordova Custom Plugin - Async Request

Cordova app - Plugins

Test plugin 2 - Android plugin - part 1

- now setup and tested our initial JS only plugin application
- JS only can be a particularly useful way to develop a custom plugin
- often necessary to create one using the native SDK for a chosen platform
  - *eg: a custom Android plugin*
- now create a second test application
  - *then start building our test custom Android plugin*

```
cordova create customplugintest2 com.example.customplugintest2 customplugintest2
```

- add test template to application

Cordova app - Plugins

Test plugin 2 - Android plugin - part 2

- start to consider developing our custom Android plugin
- Android plugins are written in Java for the native SDK
- build a test plugin to help us understand process for working with native SDK
- test a few initial concepts for our plugin
  - *processing user input,*
  - *returning some output to the user*
  - *some initial error handling*

Cordova app - Plugins

Test plugin 2 - Android plugin - part 3

- now consider setup of our application to help us develop a native Android plugin
- three parts to a plugin that need concern us as developers

```
|- plugin.xml
|- src
   |- android
      |- Test2.java
|- www
   |- test2.js
```

- then add our required platforms for development

```
// add android
plugman platform add --platform_name android
```

- focus on the Android platform for the plugin

Test plugin 2 - Android plugin - part 4

- start to build our native Android plugin
- begin by modifying the `Test2.java` file
- Cordova Android plugins require some default classes

```
import org.apache.cordova.CordovaPlugin;
import org.apache.cordova.CallbackContext;
```

- our Java code begins importing required classes for a standard plugin
- these include Cordova required classes
  - *required for general Android plugin development*

Cordova app - Plugins

Test plugin 2 - Android plugin - part 5

- now start to build our plugin's class
- start by creating our class, which will extend CordovaPlugin

```
public class Test2 extends CordovaPlugin {
  ...do something useful...
}
```

- then start to consider the internal logic for the plugin
- each Android based Cordova plugin requires an `execute()` method
- this method is run
  - *whenever our Cordova application requires interaction or communication with a plugin*
  - *this is where all of our logic will be run*

```
@Override
public boolean execute(String action, JSONArray args, CallbackContext callbackContext)
throws JSONException {
    if (action.equals("coolMethod")) {
        String message = args.getString(0);
        this.coolMethod(message, callbackContext);
        return true;
    }
    return false;
}
```

Test plugin 2 - Android plugin - part 6

- for the execute method
  - *passing an action string*
  - *tells plugin what is being requested*
- plugin uses this requested action
  - *checks which action is being used at a given time*
  - *eg: plugins will often have many different features*
- code within `execute()` method needs to be able to check the required action
- now update our `execute()` method,

```
@Override
public boolean execute(String action, JSONArray args, CallbackContext callbackContext)
throws JSONException {
    if (ACTION_GET_NOTE.equals(action)) {
        JSONObject arg_object = args.getJSONObject(0);
        String note = arg_object.getString("note");
    }
    String result = "Your daily note: "+note;
    callbackContext.success(result);
    return true;
}
```

Test plugin 2 - Android plugin - part 7

- with our updated `execute()` method
  - *if the request action is `getNote`*
  - *our Java code grabs requested input from JSON data structure*
- current test plugin has a single input value
- if we started to build out the plugin
  - *eg: requiring additional inputs*
  - *we could grab them from the JSON as well*
- we've also added some basic error handling
- able to leverage the default `callbackContext` object
  - *provided by the standard Cordova plugin API*
- able to simply return an error to the caller
  - *if an invalid action is requested*
- one of the good things about developing an Android plugin for Cordova
  - *majority of plugins follow a similar pattern*
  - *main differences will be seen within the `execute()` method*

Cordova app - Plugins

Test plugin 2 - Android plugin - part 8

```
package org.csteach.plugin;
import org.apache.cordova.CallbackContext;
import org.apache.cordova.CordovaPlugin;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

public class Test2 extends CordovaPlugin {

  public static final String ACTION_GET_NOTE = "dailyNote";

    @Override
    public boolean execute(String action, JSONArray args, CallbackContext callbackContext)
    throws JSONException {
        if (ACTION_GET_NOTE.equals(action)) {
            JSONObject arg_object = args.getJSONObject(0);
            String note = arg_object.getString("note");
        String result = "Your daily note: "+note;
        callbackContext.success(result);
        return true;
    }
    callbackContext.error("Invalid action requested");
    return false;
    }
}
```

Test plugin 2 - Android plugin - part 9

- need to update the JavaScript for our plugin
  - *helps us expose the API for the plugin itself*
- first thing we need to do is create a primary object for our plugin
- then use this to store the APIs needed to be able to request and use our plugin

```
var noteplugin = {
... do something useful...
}

module.exports = noteplugin;
```

- current API will support one action, our `getNote` action

```
getNote:function(note, successCallback, errorCallback) {
...again, do something useful...
}
```

Test plugin 2 - Android plugin - part 10

- communication between JavaScript and the native code in the Android plugin
  - *performed using the `cordova.exec` method*
- method is not explicitly defined within our application or plugin
- when this code is run within the context of our Cordova application
  - *the `cordova` object and the required `exec()` method become available*
  - *they are part of the default structure of a Cordova application and plugin*
- now add our `cordova.exec()` method

```
cordov.exec(
...add something useful...
);
```

Test plugin 2 - Android plugin - part II

- now pass our `exec()` method two required argument
  - *represents necessary code for success and failure*
- basically telling Cordova how to react to a given user action
- then tell Cordova which plugin is required
  - *and associated action to pass to the plugin*
- also need to pass any input to the plugin
- updated `exec()` method is as follows

```
cordova.exec(
  successCallback,
  errorCallback,
  'Test2',
  'getNote',
  [{
  "note": note
  }]
);
```

Test plugin 2 - Android plugin - part 12

- plugin's JavaScript code should now look as follows

```
var noteplugin = {

  getNote:function(note, successCallback, errorCallback) {

    cordova.exec(
      successCallback,
      errorCallback,
      'Test2',
      'getNote',
      [{
        "note": note
      }]
    );

  }
}

module.exports = noteplugin;
```

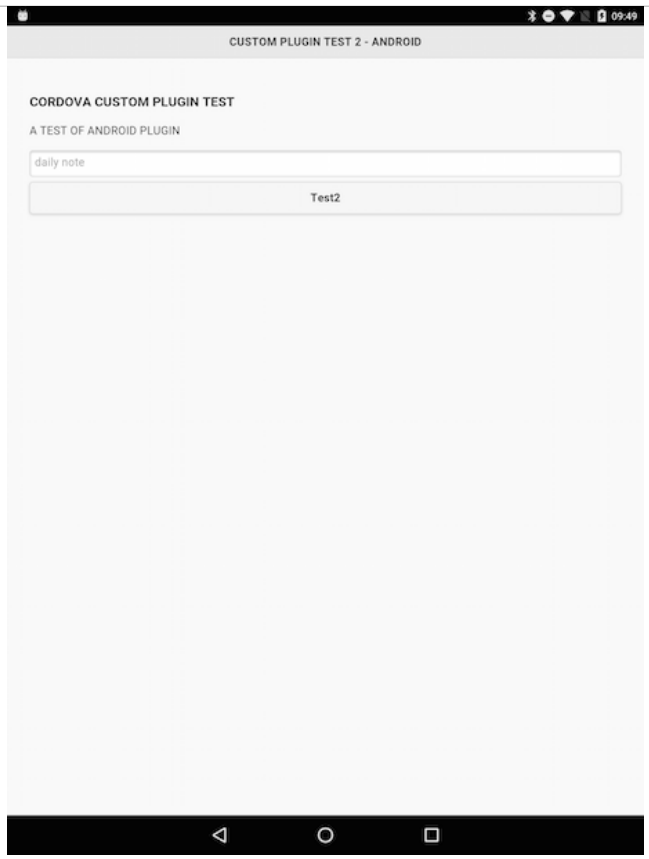
Test plugin 2 - Android plugin - part 13

- now need to test our plugin with our application
- update our home page to allow a user to interact with our new custom plugin
- add an input field for the user requested note
- add a button to submit the request itself

```
<input type="text" id="noteField" placeHolder="daily note">
<button id="testButton">Test2</button>
```

- exposed plugin API will be able to respond
  - *use the input data from the user*
  - *then pass to the native Android plugin*

Image - Cordova Custom Plugin 2



Cordova Custom Plugin 2 - HTML Update

Test plugin 2 - Android plugin - part 14

- update app's `plugin.js` to handle user input
  - *then process for use with our custom plugin*
- still need to wait for the `deviceready` event to return successfully
- then we can start to work with our user input and custom plugin
- our native Android plugin's API is similarly exposed using the window object

```
window.test2
```

- we can then execute it from our application's JS

```
windows.test2.getNote
```

- then pass the requested note data to the API
- define how we're going to work with success and error handlers
  - *render the returned value to the application's home page*

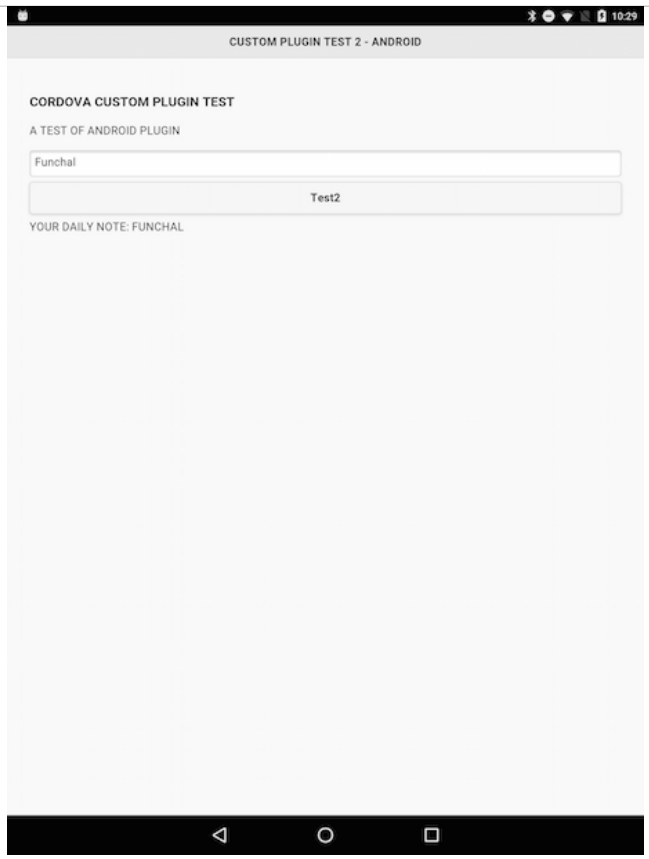
```
window.test2.getNote(note,
  function(result) {
    console.log("result = "+result);
    $("#note-output").html(result);
  },
  function(error) {
    console.log("error = "+error);
    $("#note-output").html("Note error: "+error);
  }
);
```

Cordova app - Plugins

Test plugin 2 - Android plugin - part 15

```
function onDeviceReady() {

//handle button press for daily note - direct
$("#testButton").on("tap", function(e) {
  e.preventDefault();
  console.log("request daily note...");
  var note = $("#noteField").val();
  console.log("requested note = "+note);
  if (note === "") {
    return;
  }
  window.test2.getNote(note,
    function(result) {
      console.log("result = "+result);
      $("#note-output").html(result);
    },
    function(error) {
      console.log("error = "+error);
      $("#note-output").html("Note error: "+error);
    }
  );
});

}
```

Image - Cordova Custom Plugin 2



Cordova Custom Plugin 2 - Android plugin output

Cordova app - Plugins

Summary of custom plugin development

- an initial template for a custom plugin can be created using the *Plugman* tool
- create JS only custom plugins
- create native SDK plugins
  - *eg: Android, iOS, Windows Phone...*
- custom plugin consists of
  - *plugin.xml*
  - *JavaSript API*
  - *native code*
- create the plugin separate from the application
  - *then add to an application for testing*
  - *remove to make changes, then add again...*

References

Cordova API

- config.xml
- Hooks
- Merges
- Network Information
- plugins
- plugin - file transfer
- plugin - globalization
- Plugin Development Guide
- Plugin.xml

MDN - JavaScript reference

- String.prototype.split()
- RegExp

OnsenUI

- OnsenUI v2
- JavaScript Reference