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

Fall Semester 2017 - Week 6

Dr Nick Hayward

DEV Week Assessment

- demo and project report
  - *due on Friday 20th October 2017 @ 2.45pm*
- anonymous peer review
  - *similar to user comments and feedback*
  - *chance to respond to feedback before final project*

DEV Week Assessment

Course total = 30%

- cross-platform mobile app from scratch
  - *can be basic demo of intended final app*
- build using either
  - *Apache Cordova and UI (jQuery Mobile, OnsenUI &c.)*
  - *React Native*
- can be platform agnostic (cross-platform) or specific targeted OS, e.g.
  - *cross-platform app that builds for Android and iOS*
  - *targeted build for Android or iOS*
  - *consider choice, and explain why?*
- outline concept, research conducted to date
- consider applicable design patterns
- are you using any sensors &c. ?
  - *how, why?*
- prototyping
  - *demo current prototypes*
  - *any working tests or models &c.*
- anything else to help explain your mobile app...

DEV Week Demo

DEV week assessment will include the following:

- brief presentation or demonstration of current project work
  - *due on Friday 20th October 2017*
  - *~ 10 minutes per group*
- presentation and demonstration...
  - *outline mobile app*
  - *show prototypes and designs*
  - *explain what does & does not work*
  - *...*

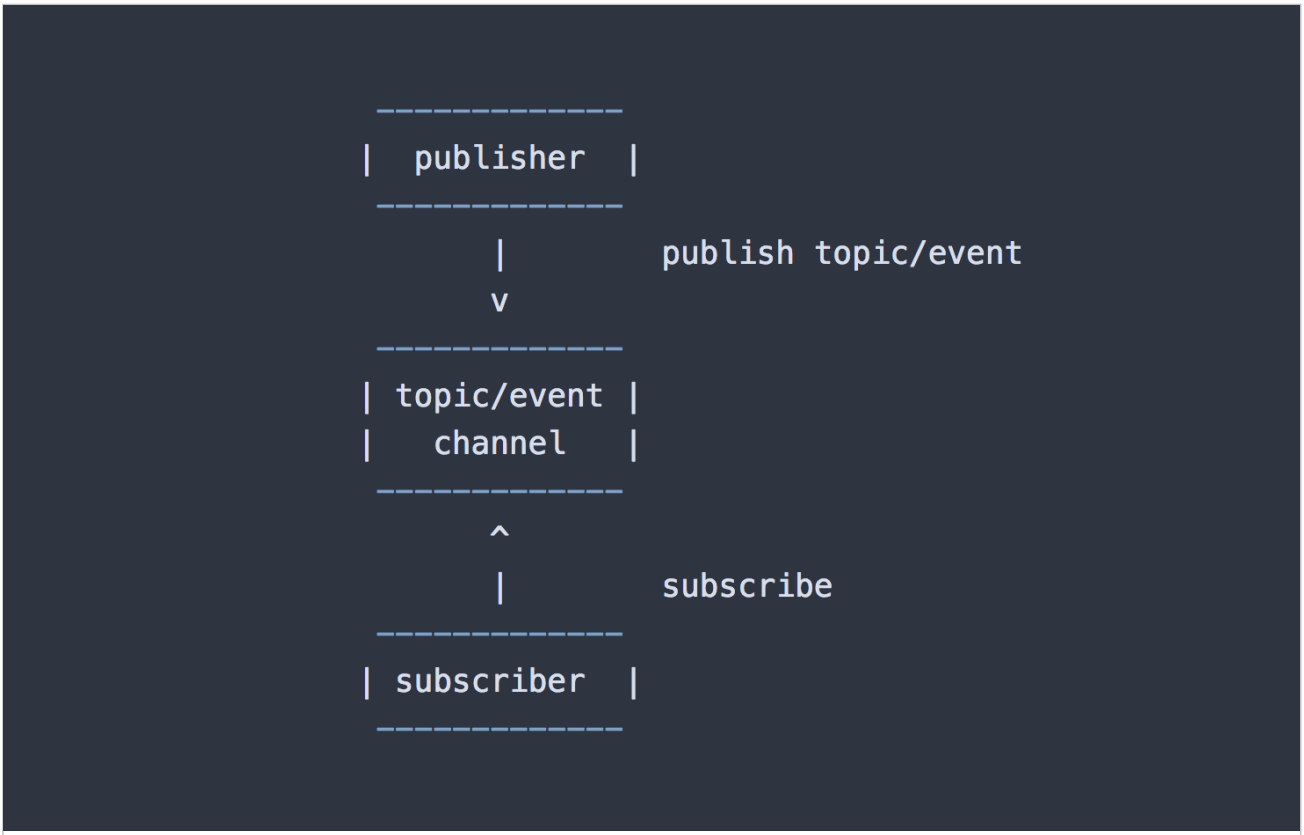
Design Patterns - Pub/Sub - intro

- variation of standard *observer* pattern is *publication and subscription*
  - *commonly known as PubSub pattern*
- popular usage in JavaScript
- *PubSub* pattern publishes a *topic* or *event* channel
- publication acts as a *mediator* or *event system* between
  - *subscriber objects wishing to receive notifications*
  - *and publisher object announcing an event*
- easy to define specific events with *event system*
- events may then pass custom arguments to a subscriber
- trying to avoid potential dependencies between objects
  - *subscriber objects and the publisher object*

Design Patterns - Pub/Sub - abstraction

- inherent to this pattern is the simple abstraction of responsibility
- publishers are unaware of nature or type of subscribers for messages
- subscribers are unaware of the specifics for a given publisher
- subscribers simply identify their interest in a given topic or event
  - *then receive notifications of updates for a given subscribed channel*
- primary difference with *observer* pattern
  - *PubSub abstracts the role of the subscriber*
- *subscriber* simply needs to handle data broadcasts by a *publisher*
- creating an abstracted *event system* between objects
  - *abstraction of concerns between publisher and subscriber*

Image - Publish/Subscribe Pattern



PubSub Pattern

Design Patterns - Pub/Sub - benefits

- *observer* and *PubSub* patterns help developers
  - *better understanding of relationships within an app's logic and structure*
- need to identify aspects of our app that contain direct relationships
- many direct relationships may be replaced with patterns
  - *subjects and observers*
  - *publishers and observers*
- tightly coupled code can quickly create issues
  - *maintenance, scale, modification, clarity of code and logic...*
  - *semmingly minor changes may often create a cascade or waterfall effect in code*
- a known side effect of tightly couple code
  - *frequent need to mock usage &c. in testing*
  - *time consuming and error prone as app scales...*
- *PubSub* helps create smaller, loosely coupled blocks
  - *helps improve management of an app*
  - *promotes code reuse*

Design Patterns - Pub/Sub - basic example - part 1 - event system

```
// constructor for pubsub object
function PubSub () {
this.pubsub = {};
}

// publish - expects topic/event & data to send
PubSub.prototype.publish = function (topic, data) {
  // check topic exists
  if (!this.pubsub[topic]){
    console.log(`publish - no topic...`);
    return false;
  }
  // loop through pubsub for specified topic - call subscriber functions...
  this.pubsub[topic].forEach(function(subscriber) {
      subscriber(data || {});
    });
};

// subscribe - expects topic/event & function to call for publish notification
PubSub.prototype.subscribe = function (topic, fn) {
  // check topic exists
  if (!this.pubsub[topic]) {
    // create topic
    this.pubsub[topic] = [];
    console.log(`pubsub topic initialised...`);
  }
  else {
    // log output for existing topic match
    console.log(`topic already initialised...`);
  }
  // push subscriber function to specified topic
  this.pubsub[topic].push(fn);
};
```

Design Patterns - Pub/Sub - basic example - part 2 - usage

```
// basic log output
var logger = data => { console.log( `logged: ${data}` ); };

// test function for subscriber
var domUpdater = function (data) {
  document.getElementById('output').innerHTML = data;
}

// instantiate object for PubSub
const pubSub = new PubSub();

// subscriber tests
pubSub.subscribe( 'test_topic', logger );
pubSub.subscribe( 'test_topic2', domUpdater );
pubSub.subscribe( 'test_topic', logger );

// publisher tests
pubSub.publish('test_topic', 'hello subscribers of test topic...');
pubSub.publish('test_topic2', 'update notification for test topic2...');
```

- Demo - Pub/Sub

Server-side considerations - data storage

Firebase - mobile platform - what is it?

- other data store and management options now available to us as developers
- depending upon app requirements consider
  - *Firebase*
  - *RethinkDB*
- as a data store, Firebase offers a hosted NoSQL database
  - *data store is JSON-based*
  - *offering quick, easy development from webview to data store*
- syncs an app's data across multiple connected devices in milliseconds
  - *available for offline usage as well*
- provides an API for accessing these JSON data stores
  - *real-time for all connected users*
- Firebase as a hosted option more than just data stores and real-time API access
- Firebase has grown a lot over the last year
  - *many new features announced at Google I/O conference in May 2016*
  - *analytics, cloud-based messaging, app authentication*
  - *file storage, test options for Android*
  - *notifications, adverts...*

Server-side considerations - data storage

RethinkDB - realtime JSON - what is it?

- RethinkDB describes itself as,

> *open source, scalable JSON database built from the ground up for the realtime web*

- RethinnkDB can be setup on a server, as a cloud service...
  - *offers flexibility, customisation, performance benefits to different teams and apps*
- paradigm shift is how an app can consume data with RethinkDB
- mobile app can now consume a continuous stream of data
  - *pushed real-time from a RethinkDB data store*
  - *create real-time, scalable apps*
- use this type of real-time model for various types of apps, e.g.
  - *gaming apps, including multi-player polling and communication*
  - *live updates for auctions, sales, and other marketplaces...*
- RethinkDB inherently different from real-time sync APIs
  - *closer to a standard database in its underlying structure, options, and general functionality*
  - *developer can use queries such as table joins, geospatial queries, subqueries...*
- build mobile apps to scale to open thousands of concurrent feeds on a single instance
- leverage clusters to enable hundreds of thousands of concurrent feeds

Server-side considerations - data storage

working with mobile cross-platform designs

- how can we use Redis, MongoDB, and other data store technologies with Cordova?
- considerations for a multi-platform structure
  - *data*
  - *models*
  - *views*
- authentication
  - *user login*
  - *accounts*
  - *data*

Data considerations in mobile apps

- worked our way through Cordova's File plugin
  - *tested local read and write for files*
- test JS requests with JSON
  - *local and remote files*
  - *remote services and APIs*
- work natively with JS objects
  - *webview*
  - *controller*
  - *local or remote data store or service*

Cordova app - ES6 Generators & Promises - intro

- generators and promises are new to plain JavaScript
  - *introduced with ES6 (ES2015)*
- **Generators** are a special type of function
  - *produce multiple values per request*
  - *suspend execution between these requests*
- *generators* are useful to help simplify convoluted loops
- suspend and resume code execution, &c.
  - *helps write simple, elegant async code*
- **Promises** are a new, built-in object
  - *help development of async code*
- promise becomes a placeholder for a value not currently available
  - *but one that will be available later*

Cordova app - ES6 Generators & Promises - async code and execution

- JS relies on a single-threaded execution model
- query a remote server using standard code execution
  - *block the UI until a response is received and various operations completed*
- we may modify our code to use callbacks
  - *invoked as a task completes*
  - *should help resolve blocking the UI*
- callbacks can quickly create a *spaghetti* mess of code, error handling, logic...
- *Generators* and *Promises*
  - *elegant solution to this mess and proliferation of code*

Cordova app - ES6 Generators & Promises - generators

- a *generator* function generates a sequence of values
  - *commonly not all at once but on a request basis*
- generator is explicitly asked for a new value
  - *returns either a value or a response of no more values*
- after producing a requested value
  - *a generator will then suspend instead of ending its execution*
  - *generator will then resume when a new value is requested*

Cordova app - ES6 Generators & Promises - generators - example

```
//generator function
function* nameGenerator() {
  yield "emma";
  yield "daisy";
  yield "rosemary";
}
```

- define a generator function by appending an *asterisk* after the keyword
  - `function* ()`
- use the `yield` keyword within the body of the generator
  - *to request and retrieve individual values*
- then consume these generated values using a standard loop
  - *or perhaps the new `for-of` loop*

Cordova app - ES6 Generators & Promises - generators - iterator object

- if we make a call to the body of the generator
  - *an iterator object will be created*
- we may now communicate with and control the generator using the iterator object

```
//generator function
function* NameGenerator() {
  yield "emma";
}
// create an iterator object
const nameIterator = NameGenerator();
```

- iterator object, `nameIterator`, exposes various methods including the `next` method

Cordova app - ES6 Generators & Promises - generators - iterator object - next()

- use `next` to control the iterator, and request its next value

```
// get a new value from the generator with the 'next' method
const name1 = nameIterator.next();
```

- `next` method executes the generator's code to the next yield expression
- it then returns an object with the value of the yield expression
  - *and a property `done` set to false if a value is still available*
- `done` boolean will switch to *true* if no value for next requested yield
- `done` is set to *true*
  - *the iterator for the generator has now finished*

Cordova app - ES6 Generators & Promises - generators - iterate over iterator object

- iterate over the iterator object
  - *return each value per available yield expression*
  - *e.g. use the `for-of` loop*

```
// iterate over iterator object
for(let iteratorItem of NameGenerator()) {
  if (iteratorItem !== null) {
    console.log("iterator item = "+iteratorItem+index);
  }
}
```

Cordova app - ES6 Generators & Promises - generators - call generator within a generator

- we may also call a generator from within another generator

```
//generator function
function* NameGenerator() {
  yield "emma";
  yield "rose";
  yield "celine";
  yield* UsernameGenerator();
  yield "yvaine";
}

function* UsernameGenerator() {
  yield "frisby67";
  yield "trilby72";
}
```

- we may then use the initial generator, `NameGenerator`, as normal

Cordova app - ES6 Generators & Promises - generator - recursive traversal of DOM

- document object model, or DOM, is tree-like structure of HTML nodes
- every node, except the root, has exactly one parent
  - *and the potential for zero or more child nodes*
- we may now use generators to help iterate over the DOM tree

```
// generator function - traverse the DOM
function* DomTraverseGenerator(htmlElem) {
  yield htmlElem;
  htmlElem = htmlElem.firstElementChild;
  // transfer iteration control to another instance of the
  // current generator - enables sub iteration...
  while (htmlElem) {
    yield* DomTraverseGenerator(htmlElem);
    htmlElem = htmlElem.nextElementSibling;
  }
}
```

- benefit to this generator-based approach for DOM traversal
  - *callbacks are not required*
- able to consume the generated sequence of nodes with a simple loop
  - *and without using callbacks*
- able to use generators to separate our code
  - *code that is producing values - e.g. HTML nodes*
  - *code consuming the sequence of generated values*

Cordova app - ES6 Generators & Promises - generator - exchange data with a generator

- also send data to a generator
- enables bi-directional communication
- a pattern might include
  - *request data*
  - *then process the data*
  - *then return an updated value when necessary to a generator*

Cordova app - ES6 Generators & Promises - generator - exchange data with a generator - example

```
// generator function - send data to generator - receive standard argument
function* MessageGenerator(data) {
  // yield a value - generator returns an intermediator calculation
  const message = yield(data);
  yield("Greetings, "+ message);
}

const messageIterator = MessageGenerator("Hello World");
const message1 = messageIterator.next();
console.log("message = "+message1.value);

const message2 = messageIterator.next("Hello again");
console.log("message = "+message2.value);
```

- first call with the `next()` method requests a new value from the generator
  - *returns initial passed argument*
  - *generator is then suspended*
- second call using `next()` will resume the generator, again requesting a new value
- second call also sends a new argument into the generator using the `next()` method
- newly passed argument value becomes the complete value for this yield
  - *replacing the previous value `Hello World`*
- we can achieve the required bi-directional communication with a generator
- use `yield` to return data from a generator
- then use iterator's `next()` method to pass data back to the generator

Cordova app - ES6 Generators & Promises - generator - detailed structure

Generators work in a detailed manner as follows,

- **suspended start**
  - *none of the generator code is executed when it first starts*
- **executing**
  - *execution either starts at the beginning or resumes where it was last suspended*
  - *state is created when the iterator's `next()` method is called*
  - *code must exist in generator for execution*
- **suspended yield**
  - *whilst executing, a generator may reach `yield`*
  - *it will then create a new object carrying the return value*
  - *it will yield this object*
  - *then suspends execution at the point of the yield...*
- **completed**
  - *a `return` statement or lack of code to execute*
  - *this will cause the generator to move to a complete state*

Cordova app - ES6 Generators & Promises - promises - intro

- a *promise* is similar to a placeholder for a value we currently do not have
  - *but we would like later*
- it's a guarantee of sorts
  - *eventually receive a result to an asynchronous request, computation, &c.*
- a result will be returned
  - *either a value or an error*
- we commonly use *promises* to fetch data from a server
  - *fetch local and remote data*
  - *fetch data from APIs*

Cordova app - ES6 Generators & Promises - promises - example

```
// use built-in Promise constructor - pass callback function with two parameters (resolve & reject)
const testPromise = new Promise((resolve, reject) => {
  resolve("test return");
  // reject("an error has occurred trying to resolve this promise...");
});

// use `then` method on promise - pass two callbacks for success and failure
testPromise.then(data => {
  // output value for promise success
  console.log("promise value = "+data);
}, err => {
  // output message for promise failure
  console.log("an error has been encountered...");
});
```

- use the built-in *Promise* constructor to create a new promise object
- then pass a function
  - *a standard arrow function in the above example*

Cordova app - ES6 Generators & Promises - promises - executor

- function for a Promise is commonly known as an *executor* function
  - *includes two parameters, `resolve` and `reject`*
- *executor* function is called immediately
  - *as the Promise object is being constructed*
- `resolve` argument is called manually
  - *when we need the promise to resolve successfully*
- second argument, `reject`, will be called if an error occurs
- uses the *promise* by calling the built-in `then` method
  - *available on the promise object*
- `then` method accepts two callback functions
  - *`success` and `failure`*
- `success` is called if the *promise* resolves successfully
- the `failure` callback is available if there is an error

Cordova app - ES6 Generators & Promises - promises - callbacks & async

- async code is useful to prevent execution blocking
  - *potential delays in the browser*
  - *e.g. as we execute long-running tasks*
- issue is often solved using *callbacks*
  - *i.e. provide a callback that's invoked when the task is completed*
- such long running tasks may result in errors
- issue with callbacks
  - *e.g. we can't use built-in constructs such as* `try-catch` *statements*

Cordova app - ES6 Generators & Promises - promises - callbacks & async - example

```
try {
  getJSON("data.json", function() {
    // handle return results...
  });
} catch (e) {
  // handle errors...
}
```

- this won't work as expected due to the code executing the callback
  - *not usually executed in the same step of the event loop*
  - *may not be in sync with the code running the long task*
- errors will usually get lost as part of this long running task
- another issue with callbacks is nesting
- a third issue is trying to run parallel callbacks
- performing a number of parallel steps becomes inherently tricky and error prone

Cordova app - ES6 Generators & Promises - promises - further details

- a *promise* starts in a pending state
  - *we know nothing about the return value*
  - *promise is often known as an unresolved promise*
- during execution
  - *if the promise's resolve function is called*
  - *the promise will move into its fulfilled state*
  - *the return value is now available*
- if there is an error or *reject* method is explicitly called
  - *the promise will simply move into a rejected state*
  - *return value is no longer available*
  - *an error now becomes available*
- either of these states
  - *the promise can now no longer switch state*
  - *i.e from rejected to fulfilled and vice-versa...*

Cordova app - ES6 Generators & Promises - promises - concept example

an example of working with a promise may be as follows

- code starts (execution is ready)
- promise is now executed and starts to run
- promise object is created
- promise continues until it resolves
  - *successful return, artificial timeout &c.*
- code for the current promise is now at an end
- promise is now resolved
  - *value is available in the promise*
- then work with resolved promise and value
  - *call `then` method on promise and returned value...*
  - *this callback is scheduled for successful resolve of the promise*
  - *this callback will always be asynchronous regardless of state of promise...*

Cordova app - ES6 Generators & Promises - promises - explicitly reject

- two standard ways to reject a promise
- e.g. explicit rejection of promise

```
const promise = new Promise((resolve, reject) => {
    reject("explicit rejection of promise");
});
```

- once the promise has been rejected
  - *an error callback will always be invoked*
  - *e.g. through the calling of the `then` method*

```
promise.then(
  () => fail("won't be called..."),
  error => pass("promise was explicitly rejected...");
);
```

- also chain a `catch` method to the `then` method
- as an alternative to the error callback. e.g.

```
promise.then(
  () => fail("won't be called..."))
  .catch(error => pass("promise was explicitly rejected..."));
```

Cordova app - ES6 Generators & Promises - promises - real-world promise - getJSON

```
// create a custom get json function
function getJSON(url) {
  // create and return a new promise
  return new Promise((resolve, reject) => {
    // create the required XMLHttpRequest object
    const request = new XMLHttpRequest();
    // initialise this new request - open
    request.open("GET", url);
    // register onload handler - called if server responds
    request.onload = function() {
      try {
        // make sure response is OK - server needs to return status 200 code...
        if (this.status === 200) {
          // try to parse json string - if success, resolve promise successfully with value
          resolve(JSON.parse(this.response));
        } else {
          // different status code, exception parsing JSON &c. - reject the promise...
          reject(this.status + " " + this.statusText);
        }
      } catch(e) {
        reject(e.message);
      }
    };

    // if error with server communication - reject the promise...
    request.onerror = function() {
      reject(this.status + " " + this.statusText);
    };

    // send the constructed request to get the JSON
    request.send();
  });
}
```

Cordova app - ES6 Generators & Promises - promises - real-world promise - usage

```
// call getJSON with required URL, then method for resolve object, and catch for error
getJSON("test.json").then(response => {
  // check return value from promise...
  response !== null ? "response obtained" : "no response";
}).catch((err) => {
  // Handle any error that occurred in any of the previous promises in the chain.
    console.log('error found = ', err); // not much to show due to return of jsonp from flickr...
});
```

Cordova app - ES6 Generators & Promises - promises - chain

- calling `then` on the returned promise creates a new *promise*
- if this promise is now resolved successfully
  - *we can then register an additional callback*
- we may now chain as many `then` methods as necessary
- create a sequence of promises
  - *each resolved &c. one after another*
- instead of creating deeply nested callbacks
  - *simply chain such methods to our initial resolved promise*
- to catch an error we may chain a final `catch` call
- to catch an error for the overall chain
  - *use the `catch` method for the overall chain*

```
getJSON().then()
.then()
.then()
.catch((err) => {
  // Handle any error that occurred in any of the previous promises in the chain.
    console.log('error found = ', err); // not much to show due to return of jsonp from flickr...
});
```

- if a failure occurs in any of the previous promises
  - *the `catch` method will be called*

Cordova app - ES6 Generators & Promises - promises - wait for multiple promises

- promises also make it easy to wait for multiple, independent asynchronous tasks
- with `Promise.all`, we may wait for a number of promises

```
// wait for a number of promises - all
Promise.all([
// call getJSON with required URL, `then` method for resolve object, and `catch` for error
getJSON("notes.json"),
getJSON("metadata.json")]).then(response => {
  // check return value from promise...response[0] = notes.json, response[1] = metadata.json &c.
  if (response[0] !== null) {
        console.log("response obtained");
    console.log("notes = ", JSON.stringify(response[0]));
    console.log("metadata = ", JSON.stringify(response[1]));
    }
}).catch((err) => {
  // Handle any error that occurred in any of the previous promises in the chain.
    console.log('error found = ', err); // not much to show due to return of jsonp from flickr...
});
```

- order of execution for tasks doesn't matter for `Promise.all`
- by using the `Promise.all` method
  - *we are simply stating that we want to wait...*
- `Promise.all` accepts an array of promises
  - *then creates a new promise*
  - *promise will resolve successfully when all passed promises resolve*
- it will reject if a single one of the passed promises fails
- return promise is an array of succeed values as responses
  - *i.e. one succeed value for each passed in promise*

Cordova app - ES6 Generators & Promises - promises - racing promises

- we may also setup competing promises
  - *with an effective prize to the first promise to resolve or reject*
  - *might be useful for querying multiple APIs, databases, &c.*

```
Promise.race(
  [
  // call getJSON with required URL, `then` method for resolve object, and `catch` for error
  getJSON("notes.json"),
  getJSON("metadata.json")]).then(response => {
    if (response !== null) {
        console.log(`response obtained - ${response} won...`);
    }
  }).catch((err) => {
    // Handle any error that occurred in any of the previous promises in the chain.
    console.log('error found = ', err); // not much to show due to return of jsonp from flickr...
  });
);
```

- method accepts an array of promises
  - *returns a completely new resolved or rejected promise*
  - *returns for the first resolved or rejected promise*

Cordova app - ES6 Generators & Promises - promises - combine generators and promises

an example usage for generators and promises,

- `async` function takes a *generator*, calls it, and creates the required *iterator*
  - *use iterator to resume generator execution as needed*
  - *declare a handle function - handles one return value from generator*
  - *one iteration of iterator*
  - *if generator result is a promise & resolves successfully - use iterator's `next` method*
  - *promise value sent back to generator*
  - *generator resumes execution*
  - *if error, promise gets rejected*
  - *error thrown to generator using iterator's `throw` method*
  - *continue generator execution until it returns `done`*
- `generator` - executes up to each `yield getJSON()`
  - *promise created for each `getJSON()` call*
  - *value is fetched async - generator is paused whilst fetching value...*
  - *control flow is returned to current invocation point in `handle` function whilst paused*
- `handle` function
  - *yielded value to `handle` function is a promise*
  - *able to use `then` and `catch` methods with promise object*
  - *registers success and error callback*
  - *execution is able to continue*

Cordova app - ES6 Generators & Promises - lots of examples

e.g.

- generator
  - *basic*
  - *basic-iterator*
  - *basic-iterator-over*
  - *basic-loop*
  - *basic-dom*
  - *basic-send-data*
  - *basic-send-data-2*
- promises
  - *basic*
  - *basic-cors-flickr*
  - *basic-xhr-local*
  - *basic-promise-all*
  - *basic-promise-race*
- generator & promise - async
  - *basic*

Cordova app - JS data options - JS data test 1

read local JSON file - jQuery deferred pattern

- jQuery provides a useful solution to the escalation of code for asynchronous development
- known as the `$.Deferred` object
  - *effectively acts as a central despatch and scheduler for our events*
- with the **deferred** object created
  - *parts of the code indicate they need to know when an event completes*
  - *whilst other parts of the code signal an event's status*
- **deferred** coordinates different activities
  - *enables us to separate how we trigger and manage events*
  - *from having to deal with their consequences*

Cordova app - JS data options - JS data test 1

read local JSON file - using deferred objects

- now update our AJAX request with **deferred** objects
- separate the asynchronous request
  - *into the initiation of the event, the AJAX request*
  - *from having to deal with its consequences, essentially processing the response*
- separation in logic
  - *no longer need a success function acting as a callback parameter to the request itself*
- now rely on `.getJSON()` call returning a **deferred** object
- function returns a restricted form of this **deferred** object
  - *known as a **promise***

```
deferredRequest = $.getJSON (
  "file.json",
  {format: "json"}
);
```

Cordova app - JS data options - JS data test 1

read local JSON file - using deferred objects

- indicate our interest in knowing when the AJAX request is complete and ready for use

```
deferredRequest.done(function(response) {
  //do something useful...
});
```

- key part of this logic is the `done()` function
- specifying a new function to execute
  - *each and every time the event is successful and returns complete*
  - *our AJAX request in this example*
- **deferred** object is able to handle the abstraction within the logic
- if the event is already complete by the time we register the callback via the `done()` function
  - *our **deferred** object will execute that callback immediately*
- if the event is not complete
  - *it will simply wait until the request is complete*

Cordova app - JS data options - JS data test 1

read local JSON file - error handling deferred objects

- also signify interest in knowing if the AJAX request fails
- instead of simply calling `done()`, we can use the `fail()` function
- still works with JSONP
  - *the request itself could fail and be the reason for the error or failure*

```
deferredRequest.fail(function() {
  //report and handle the error...
});
```

Cordova app - JS data options - JS data test 1

read local JSON file - working with deferred objects

`resolve()`

- use this method with the deferred object to change its state, effectively to complete
- as we resolve a deferred object
  - *any **doneCallbacks** added with `then()` or `done()` methods will be called*
  - *these callbacks will then be executed in the order added to the object*
  - *arguments supplied to `resolve()` method will be passed to these callbacks*

`promise()`

- useful for limiting or restricting what can be done to the deferred object

```
function returnPromise() {
  return $.Deferred().promise();
}
```

- method returns an object with a similar interface to a standard `deferred` object
  - *only has methods to allow us to attach callbacks*
  - *does not have the methods required to resolve or reject deferred object*
- restricting the usage and manipulation of the deferred object
  - *eg: offer an API or other request the option to subscribe to the deferred object*
  - ***NB:** they won't be able to resolve or reject it as standard*

Cordova app - JS data options - JS data test 1

read local JSON file - working with deferred objects

- still use the `done()` and `fail()` methods as normal
- use additional methods with these callbacks including the `then()` method
- use this method to return a new promise
  - *use to update the status and values of the deferred object*
  - *use this method to modify or update a deferred object as it is resolved, rejected, or still in use*
- can also combine promises with the `when()` method
  - *method allows us to accept many promises, then return a sort of master deferred*
- updated `deferred` object will now be resolved when all of the promises are resolved
  - *it will likewise be rejected if any of these promises fail*
- use standard `done()` method to work with results from all of the promises
  - *eg: could use this pattern to combine results from multiple JSON files*
  - *multiple layers within an API*
  - *staggered calls to paged results in a API...*

Cordova app - JS data options - JS data test 1

read local JSON file - update test app

- now start to update our test AJAX and JSON application
  - *begin by simply abstracting our code a little*

```
//get the notes JSON
function getNotes() {
    //return limited deferred promise object
    var $deferredNotesRequest = $.getJSON (
      "docs/json/madeira.json",
      {format: "json"}
    );
    return $deferredNotesRequest;
}

function buildNote(data) {
    //create each note's <p>
    var p = $("<p>");
    //add note text
    p.html(data);
    //append to DOM
    $("#note-output").append(p);
}
```

Cordova app - JS data options - JS data test 1

read local JSON file - working with a `promise`

- requesting our JSON file using `.getJSON()`
  - *we get a returned **promise** for the data*
- with a **promise** we can only use the following
  - *deferred object's method required to attach any additional handlers*
  - *or determine its state*
- our **promise** can work with
  - *`then`, `done`, `fail`, `always`...*
- our **promise** can't work with
  - *`resolve`, `reject`, `notify`...*
- one of the benefits of using **promises** is the ability to load one JSON file
  - *then wait for the results*
  - *then issue a follow-on request to another file*
  - *...*

Cordova app - JS data options - JS data test 1

read local JSON file - update test app

- add our `.when()` function to app
  - *`.when()` function accepts a deferred object*
  - *in our case a limited promise*
- then allows us to chain additional deferred functions
  - *including required `.done()` function*
- for returned data, use standard response object to get `travelNotes`
  - *then iterate over the array for each property*
  - *for each iteration, we can simply call our `buildNote` function*
  - *builds and renders required notes to the app's DOM*

```
$.when(getNotes()).done(function(response) {
  //get travelNotes
  var $travelNotes = response.travelNotes
  //process travelNotes array
  $.each($travelNotes, function(i, item) {
    if (item !== null) {
      var note = item.note;
      console.log(note);
      buildNote(note)
    }
  });
});
```

Cordova app - JS data options - JS data test 1

read local JSON file - update test app

- use this `.when()` function in a new function, called `.processNotes()`
- call our deferred promise object from an event handler...

```
function processNotes(){
  $.when(getNotes()).done(function(response) {
    //get travelNotes
    var $travelNotes = response.travelNotes
    //process travelNotes array
    $.each($travelNotes, function(i, item) {
      if (item !== null) {
        var note = item.note;
        console.log(note);
        buildNote(note)
      }
    });
    console.log("done..."+response.travelNotes[0].note);
  });
}
```

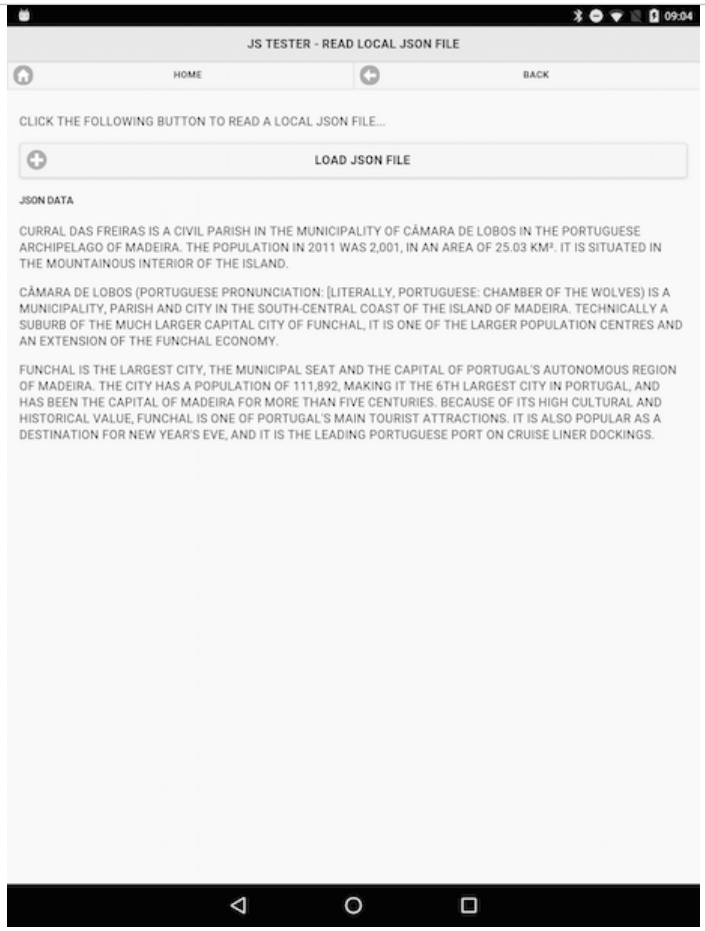
Cordova app - JS data options - JS data test 1

read local JSON file - update test app

- as we navigate to our JSON page in the test app
  - *call this function from an event handler...*

```
//handle button press for file write
$("#loadJSON").on("tap", function(e) {
  e.preventDefault();
  processNotes();
});
```

Image - API Plugin Tester - file



JS Tester - JSON deferred pattern

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

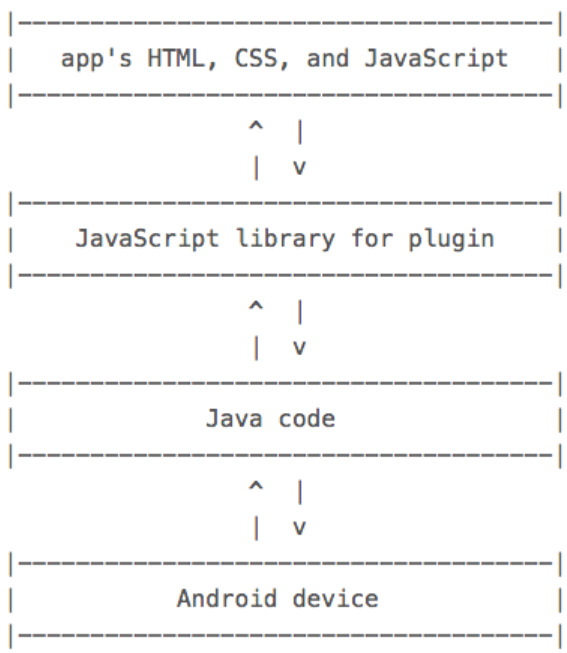
- within our plugin structure
  - *easily contain all of the required JS code in one file*
  - *divide logic and requirements into multiple files...*
- structure depends on plugin complexity and dependencies
- eg: we could bundle other jQuery plugins, handlebars.js. maps functionality...
- beyond the 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



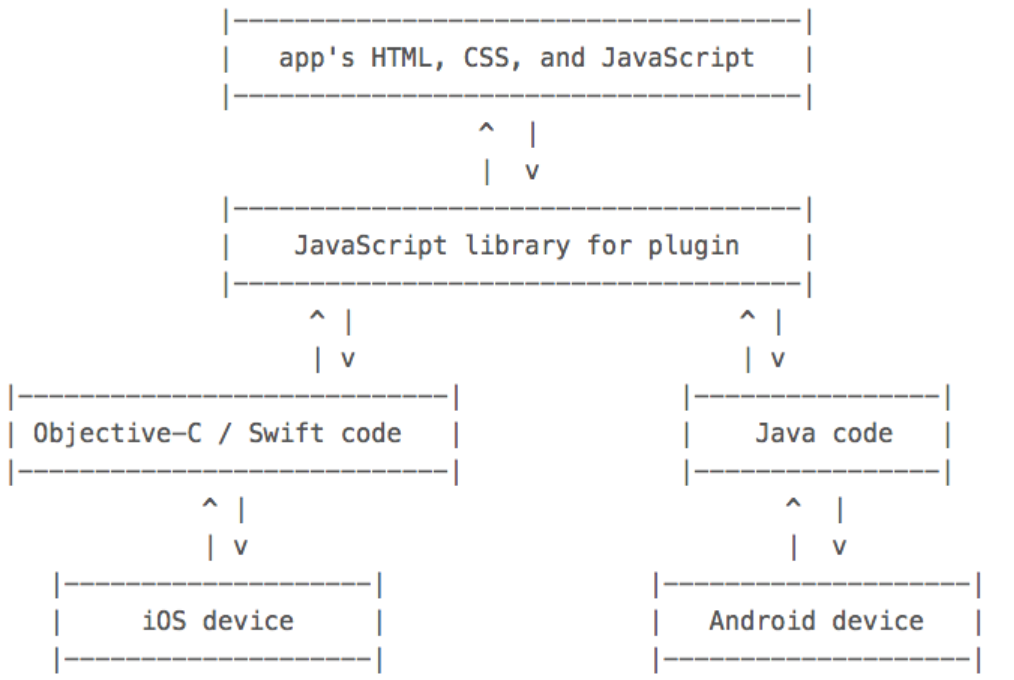
Cordova Plugin Architecture - Android

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.
```

- 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*

Cordova app - Plugins

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*

Cordova app - Plugins

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

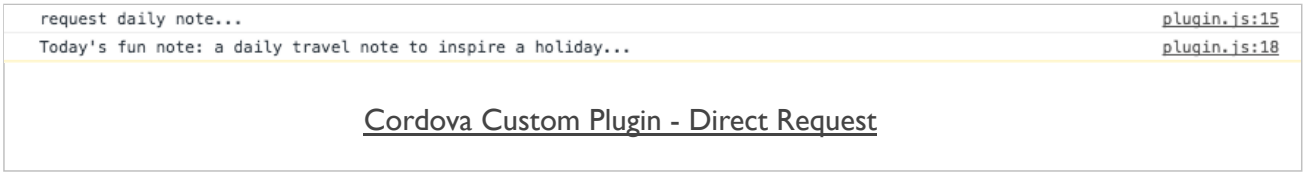
- now need to setup our home page,
- add some jQuery to handle events
- then call the exposed functions from our plugin
- start by adding some buttons to the home page

```
<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 Custom Plugin - Direct Request

Cordova app - Plugins

Test plugin 1 - JS plugin - part 9

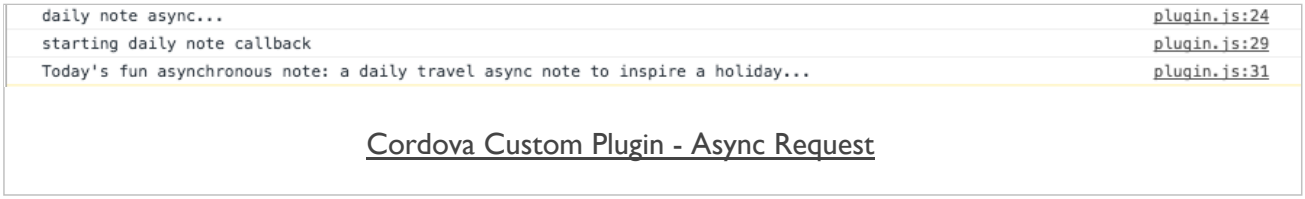
- request asynchronous version of daily note function from plugin's exposed API
- add an event handler to our `plugin.js` file
  - *responds to the request for this type of daily note*

```
//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

Cordova app - Plugins

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;
}
```

Cordova app - Plugins

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;
    }
}
```

Cordova app - Plugins

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...
);
```

Cordova app - Plugins

Test plugin 2 - Android plugin - part I I

- 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;
```

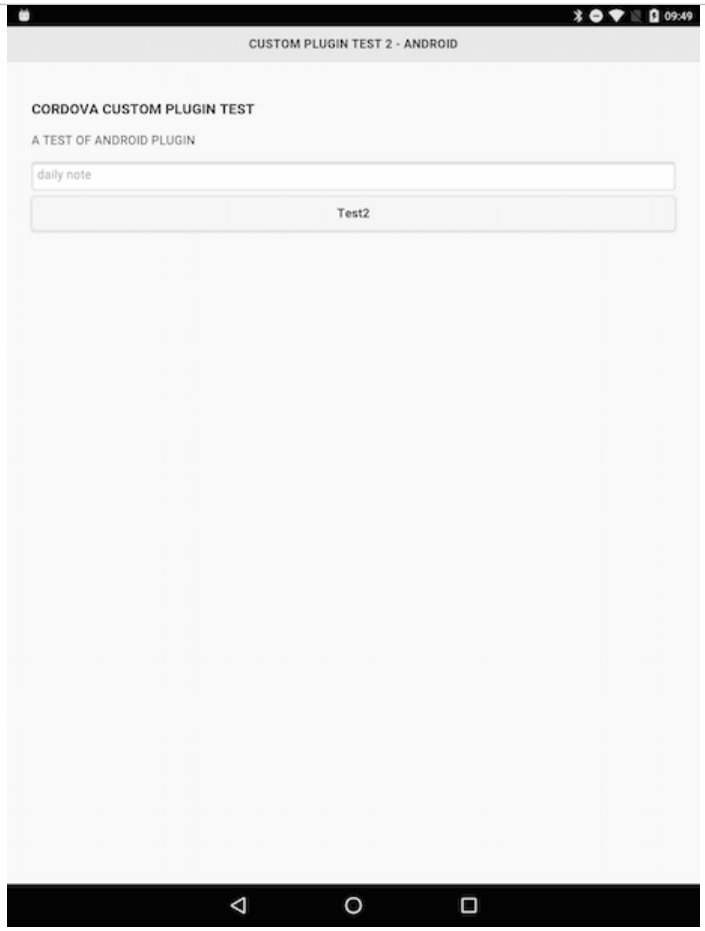
Test plugin 2 - Android plugin - part 13

- 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

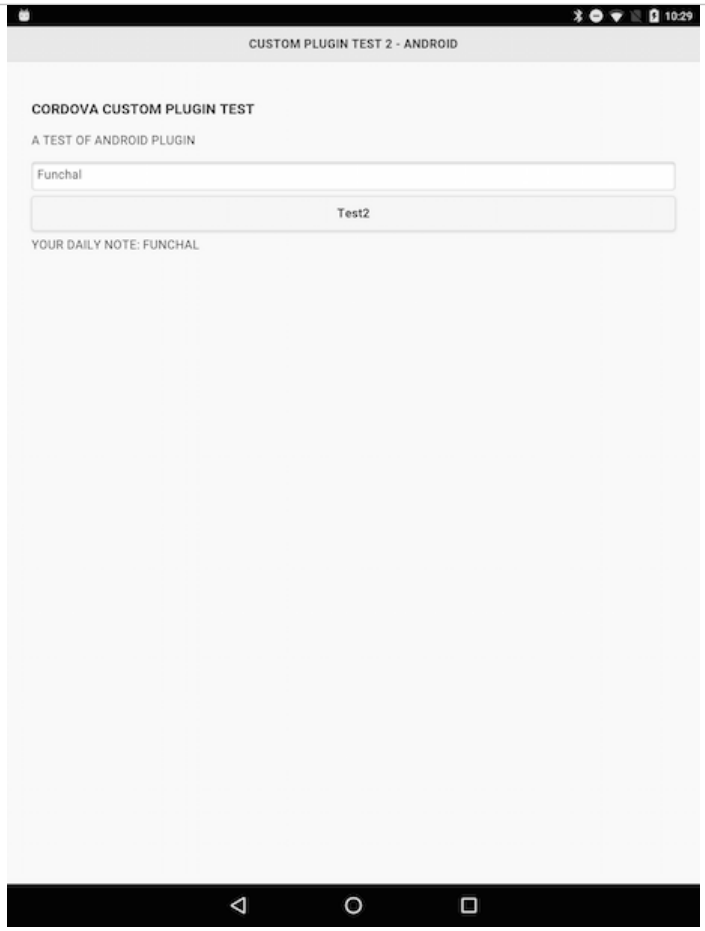
Test plugin 2 - Android plugin - part 15

```
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...*

Demos

- Pub/Sub pattern

References

- Cordova API
  - *config.xml*
  - *Hooks*
  - *Merges*
  - *Network Information*
  - *plugins*
  - *plugin - globalization*
  - *plugin - Splashscreen*
  - *plugin - statusbar*
  - *Plugin Development Guide*
  - *Plugin.xml*
- OnsenUI
  - *OnsenUI v2*
  - *JavaScript Reference*
  - *Theme Roller*