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

Fall Semester 2018 - Week 7

Dr Nick Hayward

Cross-platform JS - 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

Cross-platform JS - ES6 Generators & Promises - generators

example - pass generator to function

```
function getRandomNote(gen) {
  console.log(`getRandomNote called...`);
  const g = gen();
  fetch('./assets/input/notes.json', {
    headers: new Headers({
      Accept: 'application/json'
    })
  })
  .then(res => res.json())
  .then(json => {
      return g.next(json);
  })
  .catch(err => g.throw(err))
}

getRandomNote(function* printRandomNote() {
  console.log(`generator function executes...`);
  const json = yield;

})
```

- Demo - Generators - pass generator to function

Cross-platform JS - 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*

- traversed using depth-first search
- algorithm tries to go deeper into tree structure
  - *when it can't it moves to the next child in the list*
- e.g. define a `class` to create a Node
  - *creates with value and arbitrary amount of child nodes*

```
// Node class - holds a value and arbitrary amount of child nodes...
class Node {
  constructor(value, ...children) {
    this.value = value;
    this.children = children;
  }
}
```

Then, we create a basic node tree,

```
// define basic node tree - instantiate nodes from
const root = new Node(1,
  new Node(2),
  new Node(3,
    new Node(4,
      new Node(5,
        new Node(6)
      ),
      new Node(7)
    )
  ),
  new Node(8,
    new Node(9),
    new Node(10)
  )
)
```

- various implementations we might create for a traversal generator...

Cross-platform JS - ES6 Generators & Promises - generator function

- e.g. depth first generator function for traversing the tree

```
// FN: depthFirst generator
function* depthFirst(node) {
  yield node.value;
  for (const child of node.children) {
    yield* depthFirst(child);
  }
}

// log tree recursion
console.log([...depthFirst(root)]);
```

Cross-platform JS - 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*

Cross-platform JS - 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

Cross-platform JS - 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*

Cross-platform JS - ES6 Generators & Promises - generators & iterables

fibonacci number generator

- example generator for Fibonacci sequence
- generator will output an infinite sequence of numbers
- we may also call individual iterations of the sequence
  - *e.g.*

```
// generator function - value per iteration & done will not return true...
function* fibonacci() {
  // define start values for fibonacci sequence
  let previous = 0;
  let current = 1;
  // loop will continue to iterate fibonacci sequence
  while(true) {
    // return current value in fibonacci sequence
    yield current;
    // compute next value for sequence...
    const next = current + previous;
    // update values for next iteration of loop in fibonacci sequence
    previous = current;
    current = next;
  }
}

// instantiate iterator object using fibonacci generator
const g = fibonacci();

// call iterator
console.log(g.next());
```

- to improve performance, and prevent memory and execution timeout
  - *add **memoisation** to script*
  - *a type of local cache for the execution of the algorithm...*

Cross-platform JS - ES6 Generators & Promises - async I/O using generators

- use generators and generator helpers to create simple async input and output
  - *use with saving data &c.*
  - *a consistent and abstracted usage design for a custom generator*

```
// called with passed generator function
function saveItems(itemList) {
  const items = [];
  const g = itemList();
  return more(g.next());
  function more(item) {
    if (item.done) {
      return save(item.value);
    }
    return details(item.value);
  }
  function details(endpoint) {
    // check inputs are called & location...
    console.log(`details called - ${endpoint}`);
    return fetch(endpoint)
      .then(res => res.json())
      .then(item => {
        items.push(item);
        return more(g.next(item));
      })
  }
  function save(endpoint) {
    // check output is called & location...
    console.log(`save endpoint - ${endpoint}`);
    /*return fetch(endpoint, {
      method: 'POST',
      body: JSON.stringify({ items })
    })
    .then(res => res.json());*/
  }
}

saveProducts(function* () {
  yield './assets/input/items.json';
  yield './assets/input/notes.json';
  return './assets/output/journal.json';
})
```

Cross-platform JS - 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*

Cross-platform JS - 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*

Cross-platform JS - ES2017 Async & Await

- in ES2017, JavaScript gained native syntax to describe asynchronous operations
- now use *async/await* to work with asynchronous operations
- `Async` functions allow developers to take a promise-based implementation
  - *then use synchronous-like patterns of a generator*
  - *e.g. async implementation with sync usage patterns...*
- `await` may only be used inside `async` functions
  - *denoted with the `async` keyword*
- `async` function works in a similar manner to standard generators
  - *e.g. suspending execution in local context until a promise settles*
- if awaited expression is not originally a promise object
  - *it will be cast to a promise in this context...*

Cross-platform JS - ES2017 Async & Await - example 1

- example usage with try/catch

```
async function read() {
  // use try/catch to handle errors in awaited promises within async function
  try {
    const model = await getRandomBook();
  } catch (err) {
    console.log(err);
  }
}
// call function as usual
read();
```

- use return Promise object

```
async function read() {
  const model = await getRandomBook();
}
// call function as usual - work with return promise object...
read()
  .then()
```

Cross-platform JS - ES2017 Async & Await - example 2

```
/*
* basic-async1.js
* async called with sync-like try/catch block
* 'awaits' return from fetch to local JSON file
*/

// FN: 'fetch' from JSON
function getNotes() {
  return fetch('./assets/files/notes.json', {
    headers: new Headers({
      Accept: 'application/json'
    })
  })
  .then(res => res.json());
}

// FN: async/await
async function read() {
  try {
    const notes = await getNotes();
    console.log(`notes FETCH successful`);
  } catch (err) {
    console.log(err);
  }
}

read();
```

- Demo - Async & Await - Fetch example

Cross-platform JS - ES2017 Async & Await - example 3

initial `fetch`

```
// FN: 'fetch' from JSON
function getNotes() {
  return fetch('./assets/files/notes.json', {
    headers: new Headers({
      Accept: 'application/json'
    })
  })
  .then(res => res.json());
}
```

Cross-platform JS - ES2017 Async & Await - example 3

iterable functions

```
/*
* FNs: iterable computed data
* functions support all major ES6 data structures
* - arrays, typed arrays, maps, sets...
*/

// FN: iterable entries() - default iterator for data structure entries
function dataEntryIterator(data) {
  for (const pair of data.entries()) {
    console.log(pair);
  }
}

// FN: iterable keys() - default iterator for data structure keys
function dataKeysIterator(data) {
  for (const key of data.keys()) {
    console.log(key);
  }
}

// FN: iterable values() - default iterator for data structure values
function dataValuesIterator(data) {
  for (const value of data.values()) {
    console.log(value);
  }
}
```

Cross-platform JS - ES2017 Async & Await - example 3

async and await usage - a bit of fun...

```
// FN: async/await
async function read() {
  try {
    // await return from FETCH for notes.json file
    const data = await getNotes();
    const notes = data['notes'];
    // wrap return notes array in iterator
    const iter = notes[Symbol.iterator]();
    // test iterator with next for each result...
    console.log(iter.next());
    console.log(iter.next());
    console.log(iter.next());
    console.log(iter.next());
    console.log(`notes FETCH successful`);
    dataEntryIterator(notes);
    dataKeysIterator(notes);
    dataValuesIterator(notes);
  } catch (err) {
    console.log(err);
  }
}

read();
```

- Demo - Async & Await - example with iterables

Mobile Design & Development - Async Usage

Fun Exercise

Four groups, one app per group:

- Colours - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/colours/
- Surfing - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/surfing/
- Taxi - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/taxi/
- Trips - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/trips/

For your assigned app, consider the following

- where are **async** patterns being used within the app?
  - *consider from the perspective of a developer*
- how are these patterns being used to aid the UI design of the app?
- how is the UX of the app improved with these async patterns?

~ 10 minutes

jQuery - 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*

jQuery- 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"}
);
```

jQuery - 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*

jQuery - 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...
});
```

jQuery - 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*

jQuery - 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...*

jQuery - 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);
}
```

jQuery - 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*
  - ...

jQuery - JS data options - JS data test I

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)
    }
  });
});
```

jQuery - JS data options - JS data test I

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);
  });
}
```

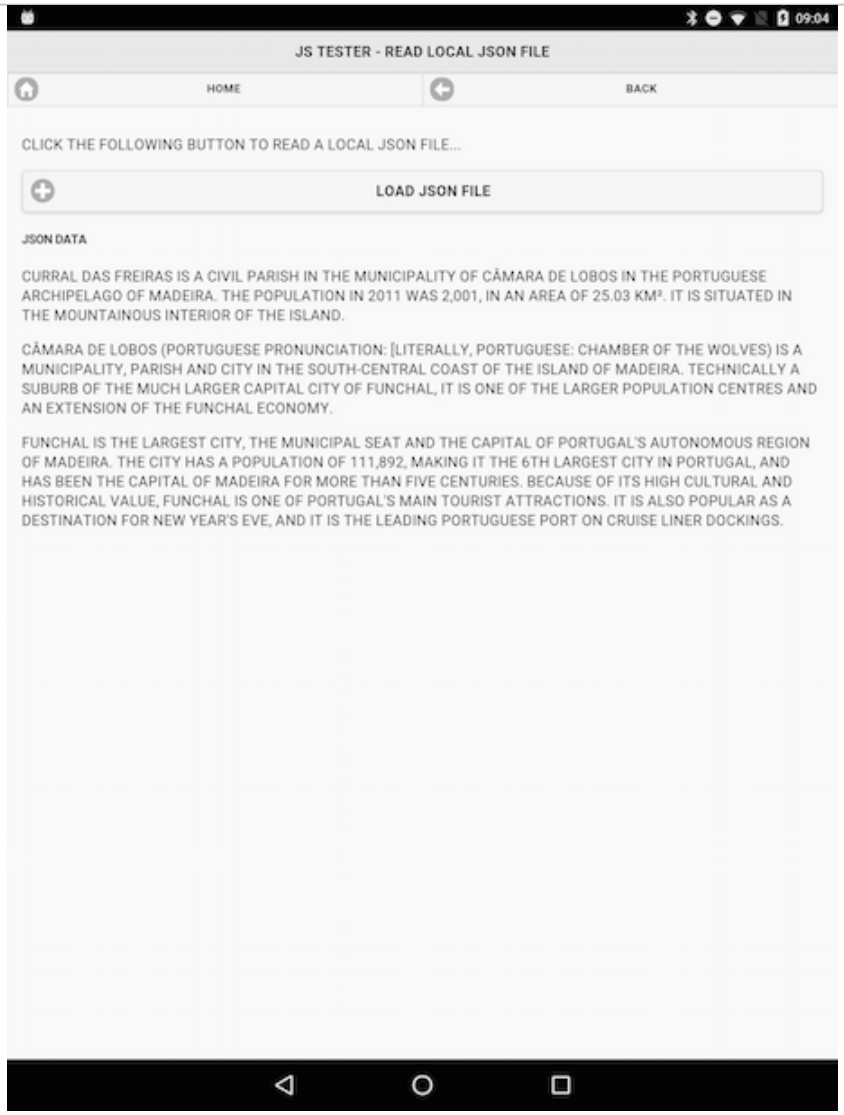
jQuery - JS data options - JS data test 1

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

References

- Cordova API
  - *Plugin Development Guide*
  - *Plugin.xml*
- Cordova Plugins
  - *Statusbar plugin*
- Google Dev
  - *Async functions*
- MDN
  - *Async function*
  - *Await*
  - *Generator*
- OnsenUI
  - *OnsenUI v2*
  - *JavaScript Reference*
  - *Theme Roller*
- Norman, D. *The Design of Everyday Things*. Basic Books. 2013.