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

Fall Semester 2018 - Week 6

Dr Nick Hayward

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*

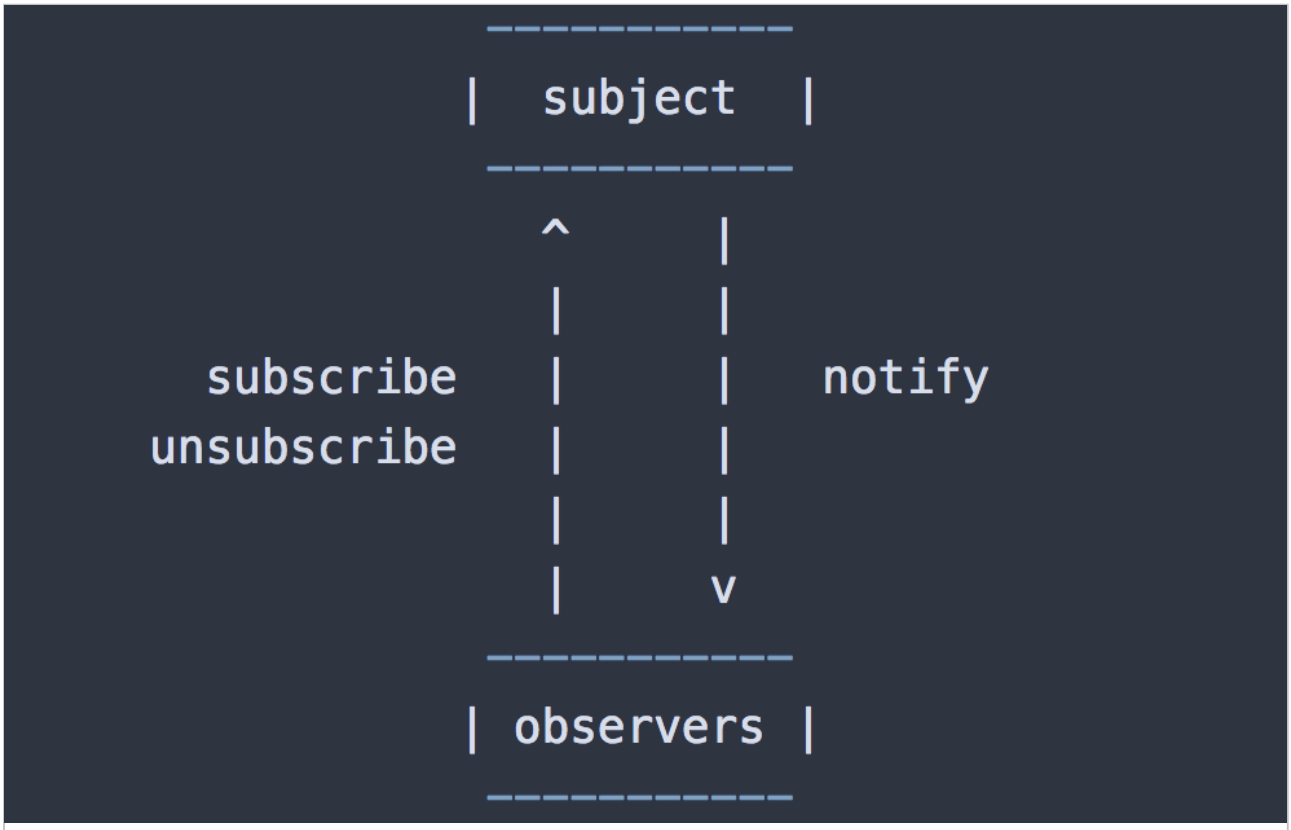
Data considerations in mobile apps

- 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*

Design Patterns - Observer - intro

- *observer* pattern is used to help define a *one to many* dependency between objects
- as **subject** (object) changes state
  - *any dependent **observers** (object/s) are then notified automatically*
  - *and then may update accordingly*
- managing changes in state to keep app in sync
- creating bindings that are event driven
  - *instead of standard push/pull*
- standard usage for this pattern with bindings
  - *one to many*
  - *one way*
  - *commonly event driven*

Image - Observer Pattern



Observer Pattern

Design Patterns - Observer - notifications

- observer pattern creates a model of event subscription with notifications
- benefit of this pattern
  - *tends to promote loose coupling in component design and development*
- pattern is used a lot in JavaScript based applications
  - *user events are a common example of this usage*
- pattern may also be referenced as *Pub/Sub*
  - *there are differences between these patterns - be careful...*

Design Patterns - Observer - Usage

The observer pattern includes two primary objects,

- **subject**
  - *provides interface for observers to subscribe and unsubscribe*
  - *sends notifications to observers for changes in state*
  - *maintains record of subscribed observers*
  - *e.g. a click in the UI*
- **observer**
  - *includes a function to respond to subject notifications*
  - *e.g. a handler for the click*

Design Patterns - Observer - Example

```
// constructor for subject
function Subject () {
  // keep track of observers
  this.observers = [];
}

// add subscribe to constructor prototype
Subject.prototype.subscribe = function(fn) {
  this.observers.push(fn);
};

// add unsubscribe to constructor prototype
Subject.prototype.unsubscribe = function(fn) {
  // ...
};

// add broadcast to constructor prototype
Subject.prototype.broadcast = function(status) {
  // each subscriber function called in response to state change...
  this.observers.forEach((subscriber) => subscriber(status));
};

// instantiate subject object
const domSubject = new Subject();

// subscribe & define function to call when broadcast message is sent
domSubject.subscribe((status) => {
  // check dom load
  let domCheck = status === true ? `dom loaded = ${status}` : `dom still loading...`;
  // log dom check
  console.log(domCheck)
});

document.addEventListener('DOMContentLoaded', () => domSubject.broadcast(true));
```

Design Patterns - Observer - Example

- Observer - Broadcast, Subscribe, & Unsubscribe

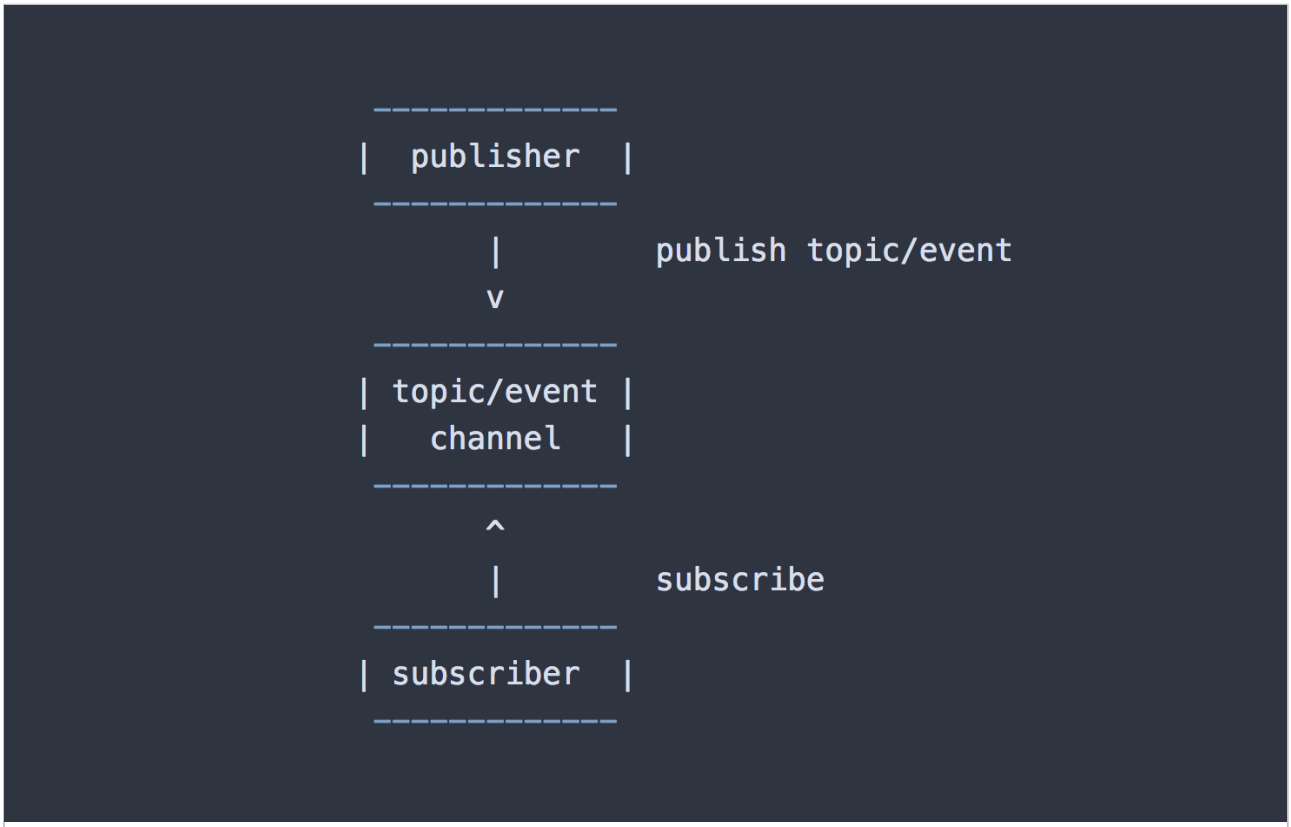
Design Patterns - Pub/Sub - intro

- 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

Mobile Design & Development - Patterns

Fun Exercise

Four groups, one app per group:

- Fast Food - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/fastfood/
- Ingredients - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/ingredients/
- Street Food - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/street-food/
- Supermarkets - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/supermarkets/

For your assigned app, consider the following

- where may you use either the Observer or Pub/Sub pattern in the app?
  - *consider from a developer's perspective*
- which parts of either pattern, Observer or Pub/Sub, creates a unified UX?
  - *consider UX in the app, and then compare with use of chosen pattern...*

~ 10 minutes

Cross-platform JS - 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*

Cross-platform JS - 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*

Cross-platform JS - 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*

Cross-platform JS - ES6 Generators & Promises - promises - example

```
// use built-in Promise constructor - pass callback function with two parameters (resolve
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*

Cross-platform JS - 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

Cross-platform JS - ES6 Generators & Promises - promises - example

explicit use of `resolve`

```
/*
* promise1.js
* wrap Array in Promise using resolve()...
*/

let testArray = Promise.resolve(['one', 'two', 'three']);

testArray.then(value => {
console.log(value[0]);
// remove first item from array
value.shift();
// pass value to chained `then`
return value;
})
.then(value => console.log(value[0]));
```

- Demo - Promise.resolve

Cross-platform JS - 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*

Cross-platform JS - 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

Cross-platform JS - 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...*

Cross-platform JS - 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...*

Cross-platform JS - ES6 Generators & Promises - promises - callbacks & async - example

promise from scratch

```
/*
* promisefromscratch-delay.js
* create a Promise object from scratch...use delay to check usage
* promise may only be called once per execution due to delay and timeout...
*/

// check promise usage relative to timer...either timeout will cause the Promise to call a
function resolveWithDelay(delay) {
  return new Promise(function(resolve, reject) {
    // log Promise creation...
    console.log('promise created...waiting');
        // resolve promise if delay value is less than 3000
    setTimeout(function() {
      resolve(`promise resolved in ${delay} ms`);
    }, delay);
        // resolve promise if delay is greater than 3000
    setTimeout(function() {
      resolve(`promise resolved in 3000ms`);
    }, 3000);
  })
}

// fulfilled with delay of 2000 ms
resolveWithDelay(2000).then(function(value) {
  console.log(value);
});
// fulfilled with default timeout of 3000 ms
// resolveWithDelay(6000).then(function(value) {
//   console.log(value);
// });
```

- Demo - Promise from scratch

Cross-platform JS - 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..."));
```

Cross-platform JS - ES6 Generators & Promises - promises - example

promise error handling

```
/*
* promise-basic-error1.js
* basic example usage of promise error handling and order...
*/

Promise
  .resolve(1)
  .then(x => {
    if (x === 2) {
      console.log('val resolved as', x);
    } else {
      throw new Error('test failed...')
    }
  })
  .catch(err => console.error(err));
```

- Demo - Promise error handling with catch

Cross-platform JS - 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 valu
          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();
  });
}
```

Cross-platform JS - 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 fl
});
```

Cross-platform JS - 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 fl
});
```

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

Cross-platform JS - 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.js
  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 fl
});
```

- 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*

Cross-platform JS - 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 err
  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 fl
  });
);
```

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

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API

- MDN - Fetch API

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API - Example

basic usage

```
/*
* fetch-basic1.js
* basic example usage of Fetch API...
*/

fetch('./assets/notes.json')
  .then(response => {
    return response.json();
  })
  .then(myJSON => {
    console.log(myJSON);
  });
```

- Demo - Fetch API - basic usage

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API - Example

catching errors

```
/*
* fetch-basic-error1.js
* basic example usage of Fetch API...chain `catch` to `then` for error handling
*/

fetch('./assets/item.json')
  .then(response => {
    // reactions passed to `then` used to handle fulfillment of a promise
    return response.json();
  })
  .then(myJSON => {
    console.log(myJSON);
  })
  .catch(err => {
    // reactions passed to `catch` executed with a rejection reason...
    console.log(`error detected - ${err}`);
  });
```

- Demo - Fetch API - catching errors

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API - Example

Fetch with Promise all

```
/*
* fetch-promise-all.js
* basic example usage of Promise.all...using Fetch API
*/

Promise
  .all([
    fetch('./assets/items.json'),
    fetch('./assets/notes.json')
  ])
  .then(responses =>
    Promise.all(responses.map(res => res.json()))
  ).then (json => {
    console.log(json);
  });
```

- Demo - Fetch API - Promise all

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API - Example

Fetch with Promise race

```
/*
* fetch-promise-race.js
* basic example usage of Promise.race...using Fetch API
*/

Promise
  .race([
    fetch('./assets/items.json'),
    fetch('./assets/notes.json')
  ])
  .then(responses => {
    return responses.json()
  })
  .then(res => console.log(res));
```

- Demo - Fetch API - Promise race

Cross-platform JS - 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*

Cross-platform JS - 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*

Cross-platform JS - 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

Cross-platform JS - 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*

Cross-platform JS - 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);
  }
}
```

References

- MDN - Generator
- MDN - Promises
- Observer Pattern
- Pub/Sub Pattern