

# Comp 324/424 - Client-side Web Design

---

Spring Semester 2020 - Week 10

Dr Nick Hayward

# 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*

# 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*

# 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*

# 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*

## 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

# 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

# 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*



# 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

## 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...*

# 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...*

# 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 and
// end
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

# 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..."));
```

# 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

# 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();
  });
}
```

# 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...
});
```



# 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*

# 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*

# 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*

# ES6 Generators & Promises - promises - Fetch API

---

- MDN - Fetch API

# 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

# 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

# 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

# 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



## 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*

# 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*

# 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

# 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*

# 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);
  }
}
```

# 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

# 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

# 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.firstChild;
  // 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*



# ES6 Generators & Promises - traversal with generators

---

- 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...



# 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)]);
```

## 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*

# 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 intermediary 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

# 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*

# 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...*

# 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());*/
  }
}

saveItems(function* () {
  yield './assets/input/items.json';
  yield './assets/input/notes.json';
});
```



```
return './assets/output/journal.json';  
})
```

# 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

# 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*

# 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...*

# 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()
```

# ES2017 Async & Await - example 2

---

## Node.js and command line

- example usage with command line arguments
  - *custom Promise object*
  - *async/await with try/catch block*
  - *initial error handling*

```
/*
 * basic-error.js
 * - error handling for async...
 */

function getArgs() {
  // Node Process command Line arguments
  const args = process.argv;
  // custom Promise object with resolve and reject
  return new Promise((resolve, reject) => {
    if (args[2] === 'test') {
      resolve(args);
    } else {
      reject('no args');
    }
  });
}

async function main() {
  try {
    let data = await getArgs();
    return data;
  } catch(e) {
    throw new Error(`main failed...${e}`);
  }
}

main()
  .then(console.log)
  .catch(console.log);
```

# 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());
}
```

# ES2017 Async & Await - example 4

---

- example fetch usage

```
/*
 * 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



# ES2017 Async & Await - example 5 - part 1

---

## *sample 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);
  }
}
```

## ES2017 Async & Await - example 5 - part 2

---

*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

# Demos

---

- Fetch API
  - *basic usage*
  - *catching errors*
  - *Fetch API & Promise.all*
  - *Fetch API & Promise.race*
- Generators - plain JS
  - *Basic*
  - *Basic Iterator*
  - *Basic Iterator Over*
  - *Basic DOM Traversal*
  - *Basic Send Data*
  - *Basic Send Data 2*
  - *Pass generator to function*
- Promises - plain JS
  - *Basic*
  - *Basic CORS Flickr*
  - *Basic Promise All*
  - *Basic Race*
  - *Basic XHR Local*
  - *Promise error handling with catch*
  - *Promise from scratch*
  - *Promise.resolve*

# Resources

---

- jQuery
  - *jQuery*
  - *jQuery API*
  - *jQuery - deferred*
  - *jQuery - .getJSON()*
  - *jQuery - JSONP*
  - *jQuery - promise*
- MDN
  - *MDN - JS*
  - *MDN - JS Const*
  - *MDN - JS - Iterators and Generators*
  - *MDN - JS Objects*