# Comp 324/424 - Client-side Web Design - Slides

Spring Semester 2018

Dr Nick Hayward

# AJAX and JSON - part 1

***intro***

- AJAX is a simple way to load data
  - *often new or updated data*
  - *into a current page without having to refresh the browser window*
- common form of data for work with AJAX is JSON
- many common usage scenarios and examples for AJAX
  - *autocomplete in forms*
  - *live filtering of search queries*
  - *real-time updates for content and data streams*
- also use AJAX to help us load data behind the scenes
  - *preparing content for our users before a specific request is received*
  - *helps to speed up page responses and data load times*
- AJAX uses an asynchronous model for processing requests
- user can continue to perform various tasks, queries, and work
  - *whilst the browser itself continues to load data*
- inherent benefit of AJAX should include
  - *a more responsive site, intuitive usage and interface experience*

## AJAX and JSON - part 2

### *asynchronous model*

- traditional synchronous model normally stops a page
  - *until it has loaded and processed a requested script*
- AJAX enables a browser to request data from the server
  - *without this synchronous pause in usage*
- AJAX's **asynchronous processing model**
  - *often known as **non-blocking***
  - *allows a page to load data and process user's interactions*
- server responds with the requested data
  - *an event will be fired by the browser*
  - *event can then call a function to process the data*
  - *often JSON, XML, or simply HTML*
- browser will use an **XMLHttpRequest** object to help handle these AJAX requests
- browser will not wait for a response

## JSON and jQuery - get a file - part 1

### *initial setup*

- try some AJAX with a JSON file

```
{
  "country":"France",
  "city":"Marseille"
}
```

- save this content to our `docs/json/trips.json` file
- run on a server, local or remote
  - *browser security restrictions for JavaScript*
  - *local server such as XAMPP, Python's SimpleHTTPServer, Node.js...*

```
python -m SimpleHTTPServer 8080
```

- initially use the `getJSON()` function to test reading this content

```
$.getJSON("docs/json/trips.json", function(trip) {
  console.log(trip);
});
```

- console output is expected JSON object

```
Object { country: "France", city: "Marseille" }
```

## JSON and jQuery - get a file - part 2

***test with site***

- now use this return object to load our data as required within a site

```
//overall app logic and loader...
function loadJSON() {
  "use strict";

  $.getJSON("docs/json/trips.json", function(trip) {
    //element for trip data
    var $tripData = $("<p>");
    //add some content from json to element
    $tripData.html(trip.city + ", " + trip.country);
    //append content to .note-output section
    $(".note-output").append($tripData);
  });
};

$(document).ready(loadJSON);
```

- DEMO - AJAX 1 - AJAX - demo 1

## JSON and jQuery - get a file - part 3

***array for trips...***

Whilst the previous example is useful, for our application we obviously need to store multiple trips. So, multiple countries, multiple cities, and so on. Therefore, we need to consider working with JSON arrays. We'll update our `trips.json` file as follows to test loading cities,

```json
{
    "cities": [
        {
            "name": "Marseille",
            "region": "Provence-Alpes-Côte d'Azur"
        },
        {
            "name": "Paris",
            "region": "Île-de-France"
        }
    ]
}
```

## JSON and jQuery - get a file - part 4

***load an array for trips...***

- update JavaScript to load array and set data as required

```
//overall app logic and loader...
function loadJSON() {
  "use strict";

  $.getJSON("docs/json/trips.json", function(trips) {
    //element for trip data
    var $cityData = $("<ul>");

    //iterate over cities array - trips.cities...
    var $cities = trips.cities;
    $cities.forEach(function (item) {
      var $city = $("<li>");
      $city.html(item.name + " in the region of " + item.region);
      $cityData.append($city);
    })
    //append list to .note-output
    $(".note-output").append($cityData);
  });
};

$(document).ready(loadJSON);
```

- DEMO - AJAX 2 - AJAX - demo 2

## Ajax, JSON & jQuery - part 1

### *jQuery Deferred*

- 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*

# Ajax, JSON & jQuery - part 2

### *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"}
);
```

# Ajax, JSON & jQuery - part 3

### *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*

# Ajax, JSON & jQuery - part 4

***handling errors with 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...
});
```

## Ajax, JSON & jQuery - part 5

***example***

- add the option to read and write from a JSON file
- we'll use AJAX for these requests
- initially we can consider our application as follows
  - *read data from JSON file*
  - *load initial data to application*
- no edit features for now
- add edit features with DB

## Ajax, JSON & jQuery - part 6

### *example - JSON*

- test reading and loading JSON file and data
- ignore standard AJAX pattern
  - *passing two callbacks, success and error*
- use `deferred` and `promise`
- initial JSON for Travel Notes app

```json
{
  "travelNotes": [{
    "created": "2015-10-12T00:00:00Z",
    "note": "a note from Cannes..."
  }, {
    "created": "2015-10-13T00:00:00Z",
    "note": "a holiday note from Nice..."
  }, {
    "created": "2015-10-14T00:00:00Z",
    "note": "an autumn note from Antibes..."
  }]
}
```

# Ajax, JSON & jQuery - part 7

***example - deferred***

- start by submitting a query for the required JSON file
- then retain the deferred object we're using for tracking
- then indicate interest in knowing when AJAX request is complete

```
//load main app logic
function loadApp() {
  "use strict";

    var $deferredNotesRequest = $.getJSON (
      "docs/json/notes.json",
      {format: "json"}
    );

    $deferredNotesRequest.done(function(response) {
        console.log("tracking json...");
    });

};
$(document).ready(loadApp);
```

## Ajax, JSON & jQuery - part 8

***example - deferred***

- `done()` method is the key part
- helps us specify the required logic to execute
  - *when the request is complete*
- if the given event has already completed as callback is registered via `done()`
  - *deferred object will execute required callback immediately*
- if not, it will simply wait until request is complete
- respond to an error
  - *add `fail()` method for errors handling and reporting*

## Ajax, JSON & jQuery - part 9

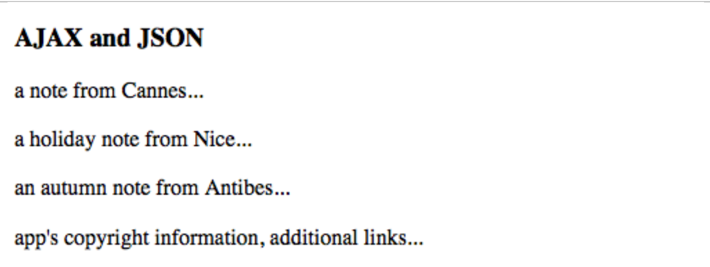
***example - work with data***

- returned data
  - *our response returns an object containing an array with notes*
- we could simply extract the required notes
  - *then append them to the DOM*

```
$deferredNotesRequest.done(function(response) {
    //get travelNotes
    var $travelNotes = response.travelNotes
    //process travelNotes array
    $travelNotes.forEach(function(item) {
      if (item !== null) {
        var note = item.note;
        //create each note's <p>
        var p = $("<p>");
        //add note text
        p.html(note);
        //append to DOM
        $(".note-output").append(p);
      }
    });
});
```

- DEMO - ajax & json basic loader

## Image - HTML5, CSS, & JS - AJAX & JSON


**AJAX and JSON**

a note from Cannes...

a holiday note from Nice...

an autumn note from Antibes...

app's copyright information, additional links...

AJAX & JSON - basic loader

# Ajax, JSON & jQuery - part 10

***example - work with data***

- we can use simple deferred requests with our local JSON data
- with staggered API calls to data, need to use slightly modified approach
  - *digging through data layer by layer*
  - *submitting a request as one layer returns*
- we could now create a second deferred object
  - *use to track additional processing requests*
  - *stagger our requests to the API*
  - *ensuring we only request certain data as needed or available*
- also create multiple deferred objects to handle our requests and returned data
  - *allows us to respond accordingly within the application*

# Ajax, JSON & jQuery - part 11

### *example - work with data*

***`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*

## Ajax, JSON & jQuery - part 12

***example - work with data***

- 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...*

## Ajax, JSON & jQuery - part 13

### *example - work with data*

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

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

//get the notes JSON
function getNotes() {
  //.get returns an object derived from a Deferred object - do not need explicit deferred object
  var $deferredNotesRequest = $.getJSON (
  "docs/json/notes.json",
  {format: "json"}
  );
  return $deferredNotesRequest;
}
```

- DEMO - ajax & json abstract loader

## Ajax, JSON & jQuery - part 14

***example - work with data***

- 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`...*

# Ajax, JSON & jQuery - part 15

## *example - work with data*

- 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*
    - ...
- a simple example of chained `then()` methods

```
getNotes().then(function(response1) {
  console.log("response1="+response1.travelNotes[2].note);
  $(".note-output").append(response1.travelNotes[2].note);
  return getPlaces();
}).then(function(response2) {
  console.log("response2="+response2.travelPlaces[2].place);
  $(".note-output").append(response2.travelPlaces[2].place);
});
```

- outputting a limited test result to the DOM and the console
- as we chain our `then()` methods
    - *pass returned results to next chained `then()` method...*
- DEMO - ajax & json deferred .then()

## 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 - 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*
- Demo - Generators - Basic

## 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*
- Demo - Generators - Basic Iterator

## 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);
  }
}
```

- Demo - Generators - Basic Iterator Over

## 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 - 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*
- Demo - Generators - Basic DOM Traversal

## 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

```javascript
// 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
- Demo - Generators - Basic Send Data
- Demo - Generators - Basic Send Data 2

## 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 - 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

```javascript
// 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*
- Demo - Promises - Basic

## 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 - 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 - 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 - 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 - 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...
});
```

- Demo - Promises - Basic XHR Local
- Demo - Promises - Basic CORS 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*
- Demo - Promises - Basic Promise All

## 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*
- Demo - Promises - Basic Race

## 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*

## Demos

- AJAX
  - *DEMO 1 - AJAX - demo 1*
  - *DEMO 2 - AJAX - demo 2*
- AJAX and JSON
  - *AJAX-JSON 1 - load a JSON file*
  - *AJAX-JSON 2 - abstract code for load a JSON file*
  - *AJAX-JSON 3 - test deferred .then()*
  - *AJAX-JSON 4 - Flickr API*
- Generators
  - *Basic*
  - *Basic Iterator*
  - *Basic Iterator Over*
  - *Basic DOM Traversal*
  - *Basic Send Data*
  - *Basic Send Data 2*
- Promises
  - *Basic*
  - *Basic XHR Local*
  - *Basic CORS Flickr*
  - *Basic Promise All*
  - *Basic Race*

## References

- Flickr API
  - *Public feeds*
  - *Public feed - public photos & video*
- jQuery
  - *jQuery*
  - *jQuery API*
  - *jQuery - deferred*
  - *jQuery - .getJSON()*
  - *jQuery - JSONP*
  - *jQuery :parent selector*
  - *jQuery - promise*
- MDN
  - *MDN - JS - Iterators and Generators*
  - *MDN - JS Objects*
  - *MDN - JS - Using Promises*