Comp 324/424 - Client-side Web Design

Spring Semester 2024 Week 11

Dr Nick Hayward

HTML5, CSS, & JS - example - part 11

add responsive design - option 2

- add new CSS - `responsive.css`
- update `index.html` with new `<link>`

```
<link rel="stylesheet" href="assets/styles/responsive.css">
```

- define initial media queries, `915px` and `545px`
- 915px for desktop to tablet
- 545px for tablet to phone &c.

HTML5, CSS, & JS - example - part 12

add responsive design - option 2

- media query for 915px
  - update wrapper, banner
  - adjust site-header & banner extras

```
div.wrapper {
    grid-template-rows: 80px auto 80px;
    margin: 20px 10px 0 10px;
}
div.banner {
    grid-template-rows: 80px;
    grid-template-areas:
        "site-logo site-header"
}
.site-header {
    margin-right: 0;
}
.banner-extras {
    display: none;
}
```

- need to update note controls, main content
- e.g. update flex for note card design
  - better use of available width

Image - HTML5, CSS, & JS - Media Query - 915px

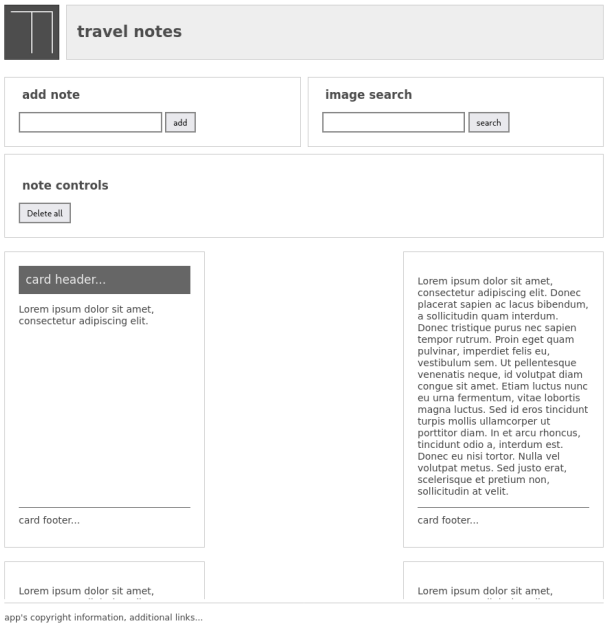

Figure 1: Media Query - 915px - banner

HTML5, CSS, & JS - example - part 13

add responsive design - option 2

- update note controls
  - move to stacked rows
  - one row per option, input field

```
.page-heading {
    grid-template-columns: 100%;
    grid-template-areas:
        "add-note"
        "search-images"
        "note-controls";
}
```

- update HTML for input group
  - container for input field and button

```
<div class="input-group">
    <input type="text" id="input-note" />
    <button id="add-note">add</button>
</div>
```

- update CSS with flex for input group

```
.input-group {
    display: flex;
    justify-content: space-around;
```

```
    flex-grow: 1;
}
```

Image - HTML5, CSS, & JS - Media Query - 915px

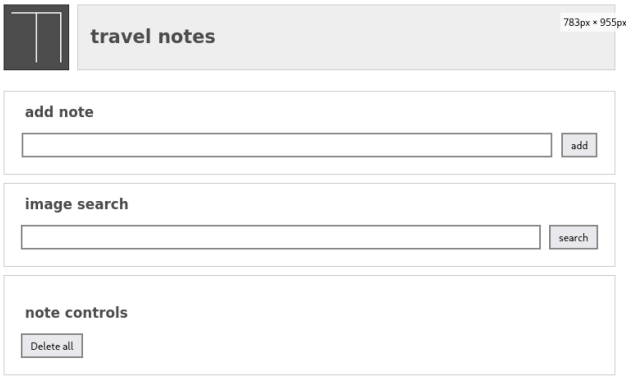

Figure 2: Media Query - 915px - note controls & input

HTML5, CSS, & JS - example - part 14

add responsive design - option 2

- update input group with modified styles, aesthetics, same sizes &c.
- add ruleset for `input-group` to style.css only
  - style applied regardless of page width

```
/* group input and button */
.input-group {
    display: flex;
    justify-content: space-around;
    flex-grow: 1;
}
/* input field */
input {
  width: 90%; /* redo with flex to fit space... */
    margin-right: 10px;
  border: 2px solid #888;
    padding: 5px;
}
```

Image - HTML5, CSS, & JS - Media Query - 915px

HTML5, CSS, & JS - example - part 15

add responsive design - option 2

- update card display to fit media query size

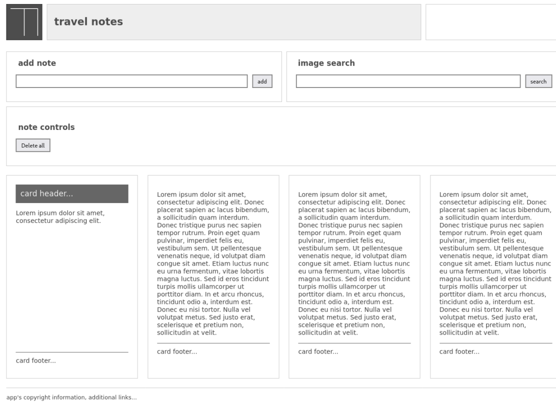

Figure 3: Media Query - 915px - input group - desktop

- many options with flex to structure cards
  - min, max, auto flex-basis & widths
  - space-evenly, add note-output border &c.
  - ...

```
/* note card - flex */
.card-view {
  display: flex;
  flex-direction: column;
    flex: 1 0 auto;
    width: 300px;
    border: 1px solid #CCCCCC;
    padding: 20px;
}
```

- flex-basis checks width

Image - HTML5, CSS, & JS - Media Query - 915px

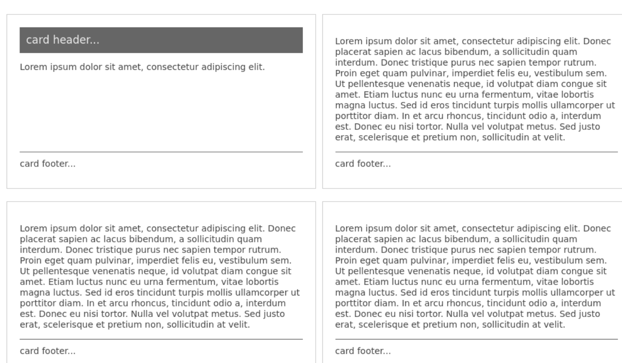

Figure 4: Media Query - 915px - note output - card layout 1

- flex-basis auto with width set to 300px

Image - HTML5, CSS, & JS - Media Query - 915px

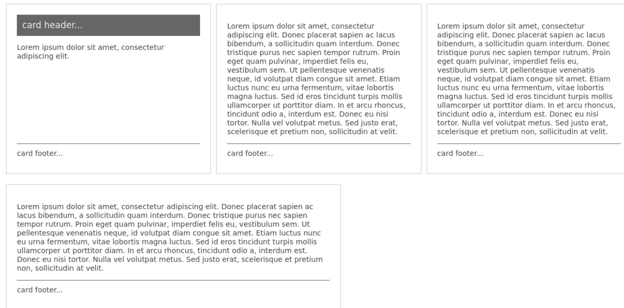

Figure 5: Media Query - 915px - note output - card layout 2

- max-width set to 50% for a note card

Image - HTML5, CSS, & JS - Media Query - 915px

- stretch to fill container

HTML5, CSS, & JS - example - part 16

add responsive design - option 2

- add media query for 545px

```
@media print, screen and (max-width: 545px) {
    ...
}
```

- update `wrapper` template rows and margins
  - improves use of screen for smaller device

```
/* grid - wrapper */
div.wrapper {
    grid-template-rows: 80px auto;
    margin: 5px 0 0 0;
}
```

- hide `note-controls` and `site-footer`
  - e.g.

```
/* hide by default */
.note-controls {
    display: none;
}
```

- DEMO - Travel Notes - Version 3 - Responsive

Image - HTML5, CSS, & JS - Media Query - 545px

- output minimalist design to phone device

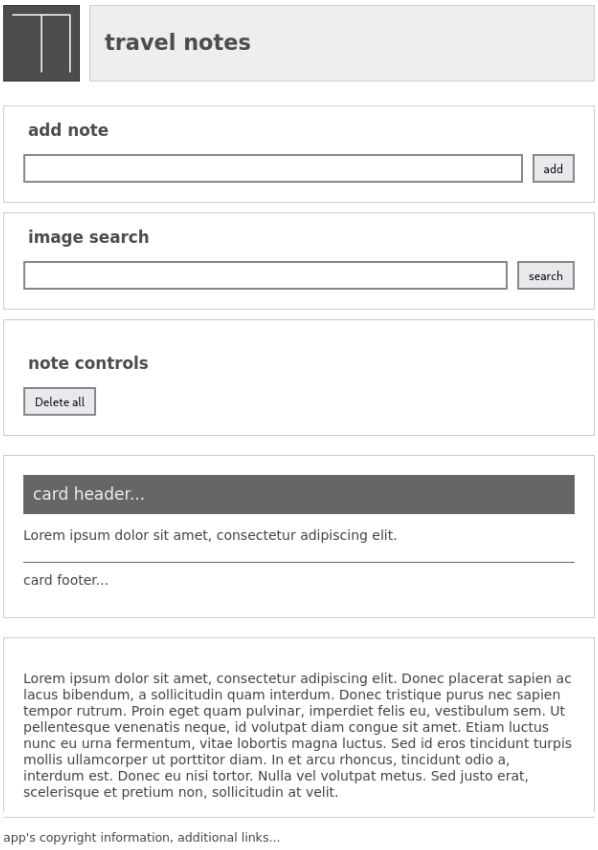

Figure 6: Media Query - 915px - note output - card layout 3

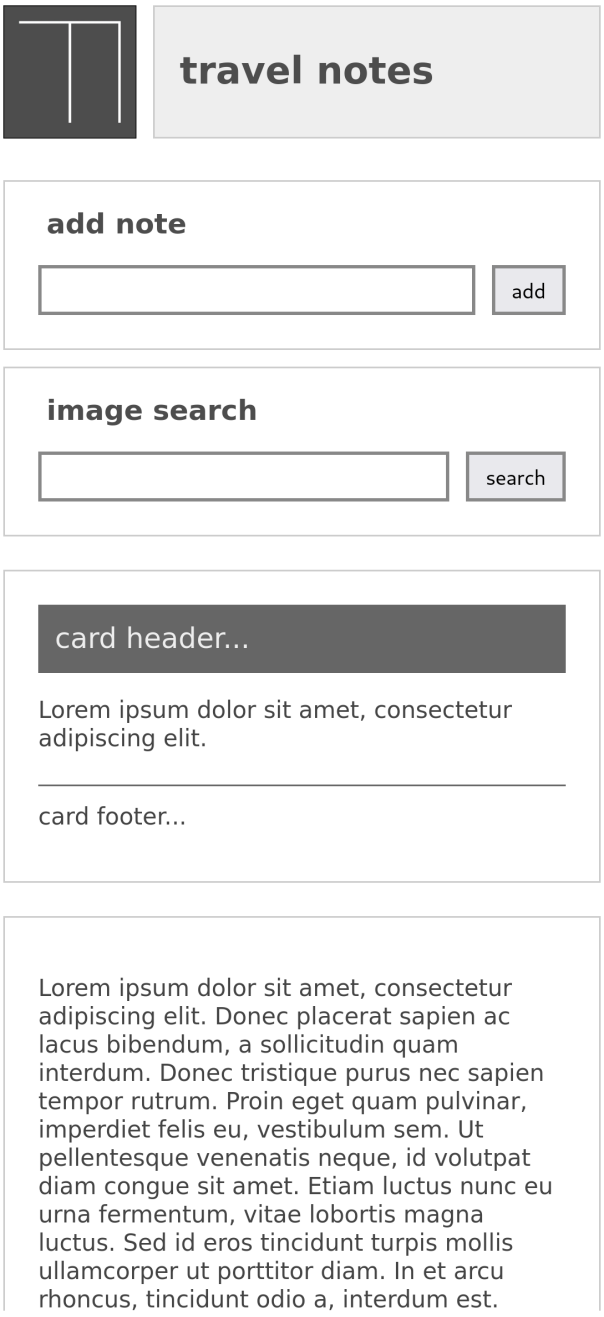

Figure 7: Media Query - 545px - phone output

CSS3 Grid - responsive layout

relative lengths

- use relative lengths and calculations for CSS property values
- for example,
  - `vw` - variable width relative to 1% of width of current viewport
  - `vh` - variable height relative to 1% of height of current viewport
  - `vmin` - relative to 1% of viewport's smaller dimension
  - `vmax` - relative to 1% of viewport's larger dimension

CSS3 Grid - responsive layout

sample updates - part 1

- after testing this type of responsive layout
  - we might add various updates
  - e.g. create a parent banner area for a header, user login, site search, and site nav

```
.banner {
    grid-area: site-banner;
    display: grid;
    grid-template-columns: auto 300px;
    grid-template-rows: 120px 60px;
    grid-template-areas:
        "site-header banner-extras"
        "site-nav site-nav";
}
```

- helps manage layout and relative sizes of banner content
  - regardless of page width and height

CSS3 Grid - responsive layout

sample updates - part 2

- `banner-extras` might be styled as follows,

```
.banner-extras {
    grid-area: banner-extras;
    display: grid;
    grid-template-areas:
        "site-user"
        "site-search";
    padding: 5%;
}
```

- use of a child grid helps us manage fixed places within the parent `banner` area

---

CSS3 Grid - responsive layout

sample updates - part 3

- update our current media query for a `min-width` of 900px

```
/* min 900 */
@media (min-width: 900px) {
    .wrapper {
        grid-template-areas:
        "site-banner site-banner"
        "content-side content"
        "site-links site-footer";
        height: 98vh;
        grid-template-columns: 250px 3fr;
        grid-template-rows: 180px auto 60px;
    }
}
```

- demo - responsive layout - part 1
- demo - responsive layout - part 2

CSS3 Grid - auto placement

dynamic content and media - part 1

- also use CSS grid with Flexbox to create content layouts
  - e.g. similar to placing cards in the UI
- we might create a layout to dynamically render images for a photo album
  - or a series of products in a brochure &c.
- start by defining a simple list with various list items

```
<ul class="items">
    <li>One</li>
    <li>Two</li>
    <li>Three</li>
    <li>Four</li>
    <li>Five</li>
    <li>Six</li>
    <li>Seven</li>
    <li>Eight</li>
    <li>Nine</li>
</ul>
```

CSS3 Grid - auto placement

dynamic content and media - part 2

- then render these list items in flexible columns within our grid layout
  - define a minimum size
  - then ensure they expand to equally fill available space
- ensures rendered layout includes equal width columns regardless of available content

```
/* content items */
.items {
    display: grid;
    grid-gap: 5px;
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
```

```
    list-style: none;
}
```

- and render individual items using flexbox

```
.items li {
    border: 1px solid #3b8eb4;
  display: flex;
  flex-direction: column;
}
```

- demo - dynamic content - part 1
- demo - dynamic content - part 2

Video - Flexible Design

fun designs Fun examples of responsive design - UP TO 0:51

Source - Example Responsive UI Designs - YouTube

CSS3 Grids - fun layout

a game board

- also use a grid layout for internal uses
  - e.g. design a game board
- basic HTML list use for 3x3 game board
  - each list item as a square on the board

```
<main class="content">
    <ul class="items">
        <li>
            <h5>One</h5>
        </li>
        <li>
            <h5>Two</h5>
        </li>
        <li>
            <h5>Three</h5>
        </li>
        <li>
            <h5>Four</h5>
        </li>
        <li>
            <h5>Five</h5>
        </li>
        <li>
            <h5>Six</h5>
        </li>
        <li>
            <h5>Seven</h5>
        </li>
        <li>
            <h5>Eight</h5>
```

```
        </li>
        <li>
            <h5>Nine</h5>
        </li>
    </ul>
</main>
```

CSS3 Grids - fun layout

a game board - part 2

- then create the grid for the `content` class

```
/* CONTENT */
.content {
    grid-area: content;
    display: grid;
    grid-template-areas:
        "items";
    grid-template-columns: 1fr;
    align-self: center;
    justify-self: center;
    align-items: center;
    padding: 50px;
    border: 1px solid #aaa;
}
```

- we can embed this content area within other grids
  - then add child items for the grid content itself
- `content` container will be aligned and justified to the centre of the parent
- each child column will occupy the same proportion of available space
  - using `grid-template-columns: 1fr`
- each item will also be aligned to the centre of the available space
- properties such as padding and border are optional
  - e.g. dictated by aesthetic requirements…

CSS3 Grids - fun layout

grid items for the board - part 1

- each square will be a child list item to the parent `ul`
  - e.g. style `ul` as follows

```
.items {
    grid-area: items;
    display: grid;
    grid-gap: 10px;
    grid-template-columns: repeat(3, 150px);
  grid-template-rows: 150px 150px 150px;
}
```

CSS3 Grids - fun layout

grid items for the board - part 2

- then style each item, which creates the squares on the game board

```
.items li {
    margin: 0;
    list-style-type: none;
    border: 1px solid #333;
    background-color: #333;
    color: #fff;
    padding: 10%;
}
```

- styling is for aesthetic purposes
  - e.g. to render a list item as a square without the default list style
- also define an alternating colour scheme for our squares, e.g.

```
.items li:nth-child(even) {
    border: 1px solid #ccc;
    background-color: #ccc;
    color: #333;
}
```

- demo - Fun with Squares

JS extras - JSON - part 1

- JSON is a lightweight format and wrapper for storing and transporting data
- inherently language agnostic, easy to read and understand
- growing rapidly in popularity
  - many online APIs have updated XML to JSON for data exchange
- syntax of JSON is itself derived from JS object notation
  - text-only format
- allows us to easily write, describe, and manipulate JSON in practically any programming language
- **JSON syntax** follows a few basic rules,
  - data is recorded as name/value pairs
  - data is separated by commas
  - objects are defined by a start and end curly brace
    - `{}`
  - arrays are defined by a start and end square bracket
    - `[]`

JS extras - JSON - part 2

- underlying construct for JSON is a pairing of name and value

```
"city":"Marseille"
```

JSON Objects

- contained within curly braces
- objects can contain multiple name/value pairs

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

JS extras - JSON - part 3

JSON Arrays

- contained within square brackets
  - arrays can also contain objects

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

- use this with JavaScript, and parse the JSON object.
  - JSFiddle - Parse JSON

JS and JSON - functions

- creating some JSON string is easy enough
- also easily create a JSON string from a JavaScript object
  - and vice-versa
- use the JavaScript `stringify` function

```
var jsonObject1 = JSON.stringify(object1);
console.log(jsonObject1);
```

- similarly parse a JSON string to a JS object

```
var object2 = JSON.parse(jsonObject1);
console.log(object2);
```

HTML5, CSS, & JS - example - part 1

add AJAX and JSON - load notes from json

- update our **travel notes** app to allow us to load some test persistent notes from a local JSON file
- initial JSON is as follows
  - modified later for specific metadata requirements

```
{
  "notes": [{
```

```
    "created": "2019-10-12T00:00:00Z",
    "note": "a note from Cannes..."
  }, {
    "created": "2021-10-13T00:00:00Z",
    "note": "a holiday note from Nice..."
  }, {
    "created": "2022-10-14T00:00:00Z",
    "note": "an autumn note from Antibes..."
  }]
}
```

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

Video - HTML5, CSS, & JS

AJAX requests AJAX requests & Twitter - UP TO 2:55

Source - What is AJAX? - YouTube

AJAX and JSON - part 3

initial usage

- try some AJAX with a JSON file

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

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

```
python3 -m http.server 4040
```

AJAX and JSON - part 4

initial usage

- initial usage with Fetch API

```
// callback for loading local JSON notes
const loadLocalNotes = file => {
    // fetch local json - promise returned
    const data = fetch(file);
    // handle return promise object
    // - take response stream, read to completion
    // - returns resolved promise with result of parsed data as JSON...
    data.then(response => response.json())
          .then(jsonData => {
                console.log(jsonData);
                // reduce each note to node tree for dom append
                const notes = jsonData['notes'].reduce((accumulator, currentVal) => {
                    const noteContainer = document.createElement('p');
                    const noteCreated = document.createTextNode(`created: ${currentVal.created}`);
                    const noteText = document.createTextNode(currentVal.note);
                    noteContainer.appendChild(noteCreated);
                    noteContainer.appendChild(noteText);
                    accumulator.appendChild(noteContainer);
                    // node tree
                    return accumulator;
                }, document.createElement('div'));
                // add node tree to dom
                noteOutput.appendChild(notes);
            });
}
```

AJAX and JSON - part 5

initial usage

- call `loadLocalNotes()` on page load, callback &c.

```
// add listener to button - anon fn for callback, enables args...
loadButton.addEventListener( 'click', () => (loadLocalNotes('notes.json')) );
```

- DEMO - Fetch - load local JSON -v1
- DEMO - Fetch - load local JSON -v2
- DEMO - AJAX 2 - AJAX - demo 2

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

```
/*
* 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]));
```

explicit use of `resolve`

- 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

```
/*
* 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);
// });
```

promise from scratch

- 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-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));
```

promise error handling

- 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);
});
```

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);
});
```

- 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);
});
```

- 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);
  });
```

- 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

```

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

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

basic usage

- DEMO - Fetch API - basic usage

ES6 Generators & Promises - promises - Fetch API - Example

```
/*
* 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}`);
  });
```

catching errors

- DEMO - Fetch API - catching errors

ES6 Generators & Promises - promises - Fetch API - Example

```
/*
* 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);
  });
```

Fetch with Promise all

- DEMO - Fetch API - Promise all

ES6 Generators & Promises - promises - Fetch API - Example

```
/*
* 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));
```

Fetch with Promise race

- DEMO - Fetch API - Promise race

Resources

- CSS
  - MDN - CSS3 Grid
  - MDN - CSS Flexbox
  - W3 Schools - CSS Grid View
  - W3 Schools - CSS Flexbox
- JavaScript
  - MDN - Fetch API
  - MDN - JS
  - MDN - JS Objects
  - MDN - Promises
- Various
  - Example Responsive UI Designs - YouTube
  - MDN - CSS3 Grid
  - Modular UI Design - YouTube