Comp 324/424 - Client-side Web Design

Spring Semester 2024 Week 12

Dr Nick Hayward

HTML5, CSS, & JS - example - part 2

add AJAX and JSON - load notes from json

- add option to load notes from JSON as app initially loads
  - use Promise pattern
  - checks source JSON as it loads via the promise
    - then outputs the end result
- example usage with Fetch API

```
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 => {
                // use return notes to populate DOM &c.
            });
}
```

- call function as follows, e.g. from button

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

- DEMO - Async demo - Fetch - local notes

HTML5, CSS, & JS - example - part 3

add AJAX and JSON - load notes from json

- help us better manage logic of our notes from app's loading and execution
  - e.g. create two separate JS files
- our updated structure might be as follows

```
...
|- assets
   |- scripts
      |- travel.js
      |- notes.js
...
```

- we can extend this further, as needed by app features and data
  - further abstraction may be added with ES Modules

HTML5, CSS, & JS - example - part 4

add AJAX and JSON - load notes from json

- add `loadLocalNotes()` function to the app's loader
  - add necessary structure to render JSON data as notes
- e.g. chain additional options, checks and balances &c.
  - each `then` method in chain
    - check return data, build DOM nodes, return and render to DOM
- for returned data, use standard response object to get `travelNotes`
  - then iterate over array for each property
    - recursion is also an option
  - for each iteration, we can call a `createNote` function
  - builds and renders required notes to the app's DOM

HTML5, CSS, & JS - example - part 5

add AJAX and JSON - recursive create note

- createNote abstracted to work with Node tree of nth depth
  - define dynamic structure for nested DOM nodes
  - use structure to build node
  - add to DOM with updated data
- create different note structure relative to context
  - basic text note
  - text note with image
  - text note with contextual data, metadata &c.
  - …
- use recursion to build nested structure
  - add required data from return JSON
  - return build node tree for DOM

HTML5, CSS, & JS - example - part 6

add AJAX and JSON - recursive create note

- example structure for node tree

```
/*
* test data for tree generation
* - obj = elem
* - array = sub-nested structure, first obj = parent
*   - nested data to nth depth
*   - maintains hierarchy, avoiding overlaps
*/
const nodeStructure = [
    {elem: 'div', attr: 'class', attrVal: 'card-view'},
    [
        {elem: 'header', attr: 'class', attrVal: 'note-header'},
```

```
        {elem: 'h5', txtNode: 'note heading'},
    ],
    [
        {elem: 'div', attr: 'class', attrVal: 'card-content'},
        [
            {elem: 'p', txtNode: inputVal},
            [
                {elem: 'span'},
                {elem: 'a'},
            ],
        ],
    ],
    {elem: 'footer'},
];
```

- nth depth supported
  - e.g. nested `span` and `a` added to test usage
- DEMO - Travel Notes - recursive create note

Image - AJAX and JSON - Recursive Create Note

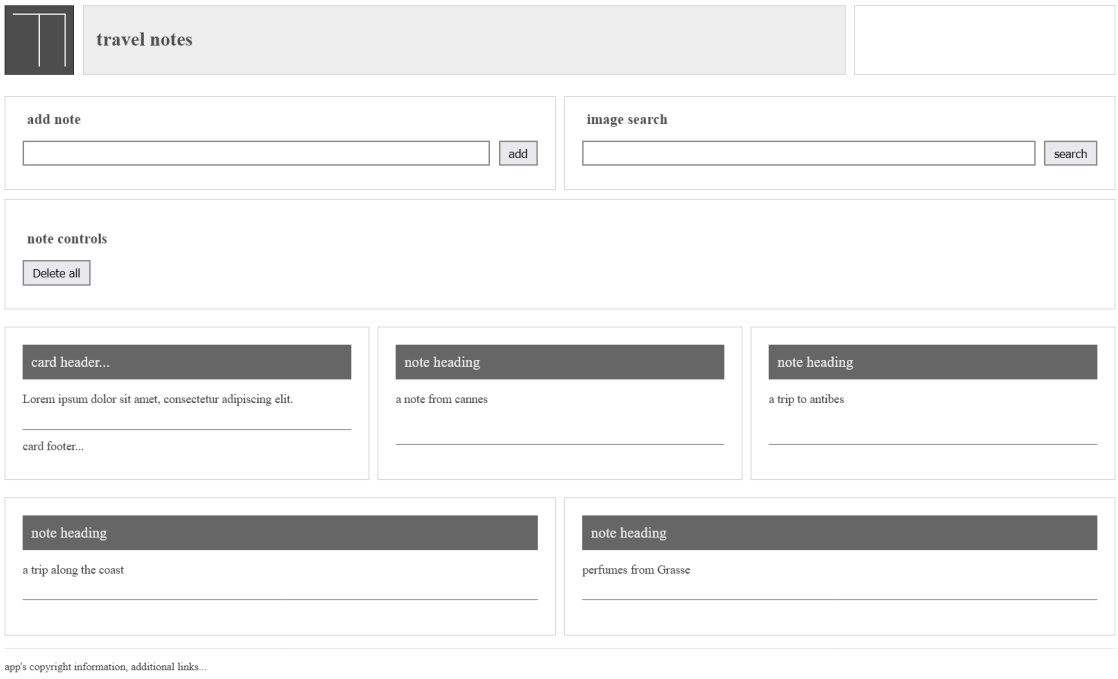

Figure 1: Travel Notes - recursive create note

HTML5, CSS, & JS - example - part 7

add AJAX and JSON - recursive create note

- define initial iteration for node structure array

```
/*
* basic check for empty string
* - extra validation, model, schema &c. required for production app
* - e.g. check for valid string &c. with regex
*/
if (inputVal !== '') {
    // build tree structure from data structure
    // - HTML DOM generation to nth depth
    const buildNodes = data => {
    let nodes = '';
    // iterate initial data - check internal objects and arrays
    for (const obj of data) {
        if (Array.isArray(obj)) {
            // check array of nodes
            // build recursive structure for DOM usage
            // ...
        } else {
            // plain object - create node for DOM usage
            const node = createNode(obj);
            // check for existing node
            if (nodes === '') {
                nodes = node;
            } else {
                // append to existing DOM nodes
                nodes.appendChild(node);
            }
        }
    }
    return nodes;
} // end of buildNodes
    // create node tree for DOM usage
    const nodeTree = buildNodes(nodeStructure);
    // add built node tree to DOM
    noteOutput.appendChild(nodeTree);
} else {
    // initial basic error handling
    console.error(`error = input data invalid`);
}
```

HTML5, CSS, & JS - example - part 8

add AJAX and JSON - recursive create note

- add function to check node structure
- `arrBuilder` called recursively to nth depth
  - check nested arrays of child nodes

```
function arrBuilder(obj) {
    const arrNodes = obj.reduce((accumulator, currentVal) => {
        //console.log(currentVal);
        if (Array.isArray(currentVal)) {
            const innerArr = arrBuilder(currentVal);
```

```
            accumulator.append(innerArr);
        } else {
            const node = createNode(currentVal);
            accumulator.appendChild(node);
        }
        return accumulator;
    }, createNode(obj.shift()));
    return arrNodes;
}
const innerNodes = arrBuilder(obj);
nodes.appendChild(innerNodes);
```

- works with variant node structures for custom tree generation
  - arrays and object combinations

HTML5, CSS, & JS - example - part 9

add AJAX and JSON - recursive create note

- full usage example

```
if (inputVal !== '') {
    // build tree structure from data structure
    // - HTML DOM generation to nth depth
    const buildNodes = data => {
    let nodes = '';
    // iterate initial data - check internal objects and arrays
    for (const obj of data) {
        if (Array.isArray(obj)) {
            function arrBuilder(obj) {
                const arrNodes = obj.reduce((accumulator, currentVal) => {
                    if (Array.isArray(currentVal)) {
                        const innerArr = arrBuilder(currentVal);
                        accumulator.append(innerArr);
                    } else {
                        const node = createNode(currentVal);
                        accumulator.appendChild(node);
                    }
                    return accumulator;
                }, createNode(obj.shift()));
                return arrNodes;
            }
            const innerNodes = arrBuilder(obj);
            nodes.appendChild(innerNodes);
        } else {
            // plain object - create node for DOM usage
            const node = createNode(obj);
            // check for existing node
            if (nodes === '') {
                nodes = node;
            } else {
                // append to existing DOM nodes
                nodes.appendChild(node);
```

```
            }
        }
    }
    return nodes;
} // end of buildNodes
    // create node tree for DOM usage
    const nodeTree = buildNodes(nodeStructure);
    // add built node tree to DOM
    noteOutput.appendChild(nodeTree);
} else {
    // initial basic error handling
    console.error(`error = input data invalid`);
}
```

HTML5, CSS, & JS - example - part 10

add AJAX and JSON - create node

- create node for passed structure
  - e.g. variant combinations of element, attribute, text node &c.

```
// create node with attrs, text node &c.
const createNode = obj => {
    const nodeElem = document.createElement(obj.elem);
    // check for attr prop in data obj
    if (obj.attr !== undefined) {
        nodeElem.setAttribute(obj.attr, obj.attrVal);
    }
    // check for txtNode prop in data obj
    if (obj.txtNode !== undefined) {
        const txtNode = document.createTextNode(obj.txtNode);
        const contentNode = nodeElem.appendChild(txtNode);
    }
    return nodeElem;
};
```

- function called in `buildNodes` to create nodes for DOM
  - nodes combined to append to app's existing tree

HTML5, CSS, & JS - example - part 11

add AJAX and JSON - create note

- `createNote` may then be called as required in app

```
/* FN: createNote
*  - input = val from input field
*  - return = DOM node for note output
*/
function createNote(input, output) {
    // get value from input field
    let inputVal = input.value;
```

```
    //  define nodeStructure

    // createNode()

    // check input value & build node tree

}
```

- call from add note button

```
// define handler for note btn click event
addNoteBtn.addEventListener('click', () => createNote(inputNote, noteOutput));
```

- DEMO - Travel Notes - recursive create note

HTML5, CSS, & JS - example - part 12

add AJAX and JSON - load notes from json

- get required DOM objects

```
// get note-output DOM object
const noteOutput = document.getElementById('note-output');
// get object for add note button
const addNoteBtn = document.getElementById('add-note');
// get input field for note
const inputNote = document.getElementById('input-note');
```

- update our event handlers for the note input button and input field keypress

```
// define handler for note btn click event
addNoteBtn.addEventListener('click', () => createNote(inputNote, noteOutput));
// define handler for note input keyboard event
inputNote.addEventListener('keypress', (e) => {
    if (e.keyCode === 13) {
        // call createNote
        createNote(inputNote, noteOutput);
        // quick clear of input field content
        inputNote.value = "";
    }
});
```

- note input button and keypress work as expected
- DEMO - Travel Notes & JSON

Image - HTML5, CSS, & JS - Travel Notes

HTML5, CSS, & JS - example - part 13

add AJAX and JSON - load notes from json

- load JSON notes by default on page load

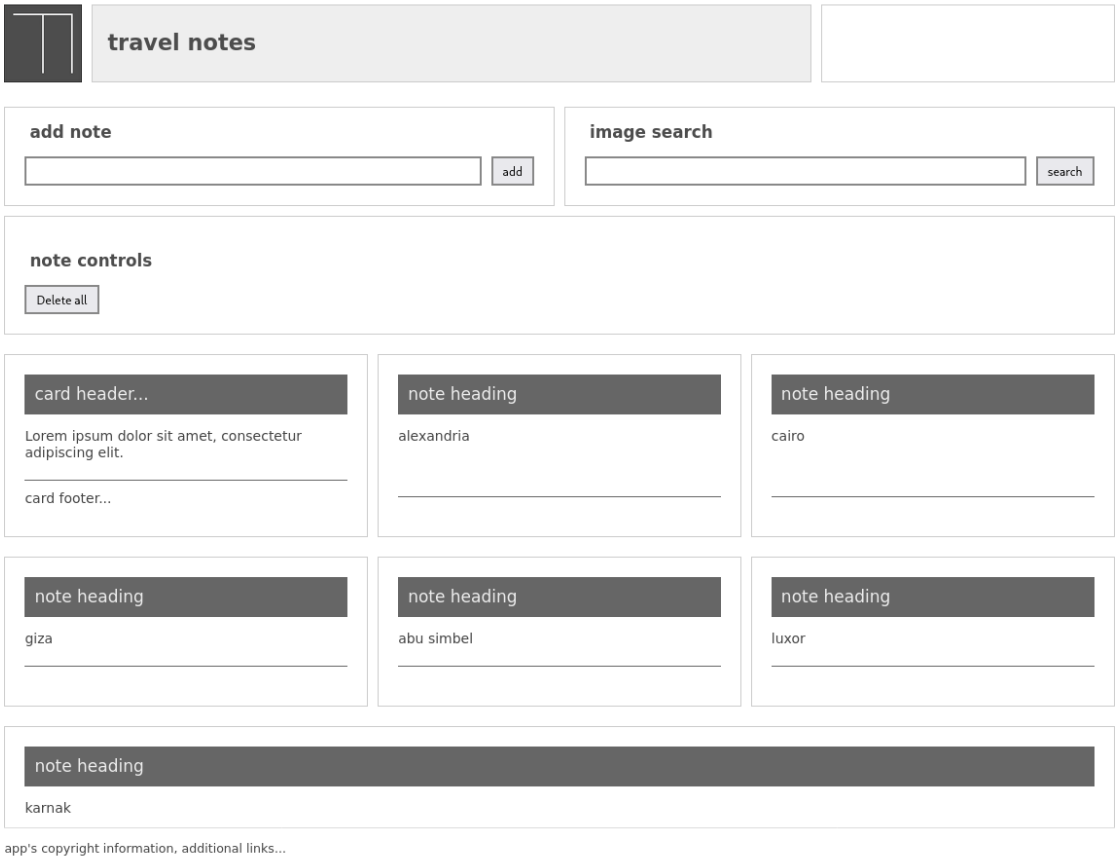

Figure 2: Travel Notes - many notes

```
const loadLocalNotes = (dir, file, output) => {
    // fetch local json, promise returned
    const data = fetch(dir + file);
    // handle return promise object
    // - take response stream, read to completion
    // - call createNote for each note object
    // - note built, added to dom
    data.then(response => response.json())
        .then(jsonData => {
            console.log(jsonData);
            // get each note object
            jsonData['notes'].map((val) => {
                // create note, output to dom
                createNote(val.note, output);
            });
        });
};
```

- call `loadLocalNotes()` on app load
  - add to `main()`

```
// get note-output DOM object
const noteOutput = document.getElementById('note-output');
// local JSON notes
const dir = './docs/';
const file = 'notes.json';
// auto-load local JSON file
loadLocalNotes(dir, file, noteOutput);
```

- DEMO - Travel Notes - auto-load notes

Image - HTML5, CSS, & JS - Travel Notes

HTML5, CSS, & JS - example - part 14

add AJAX and JSON - load notes from json

- add initial metadata to JSON note
- add extra properties as required
  - geolocation coordinates &c.

```
{
  "note": "a holiday note from Nice...",
    "metadata": {
        "created": "Tues, 19 Sep 2017 13:01:10 GMT",
        "author": "emma",
        "tags": "note, france, markets"
    }
}
```

- need to get metadata property
  - then structure for rendering

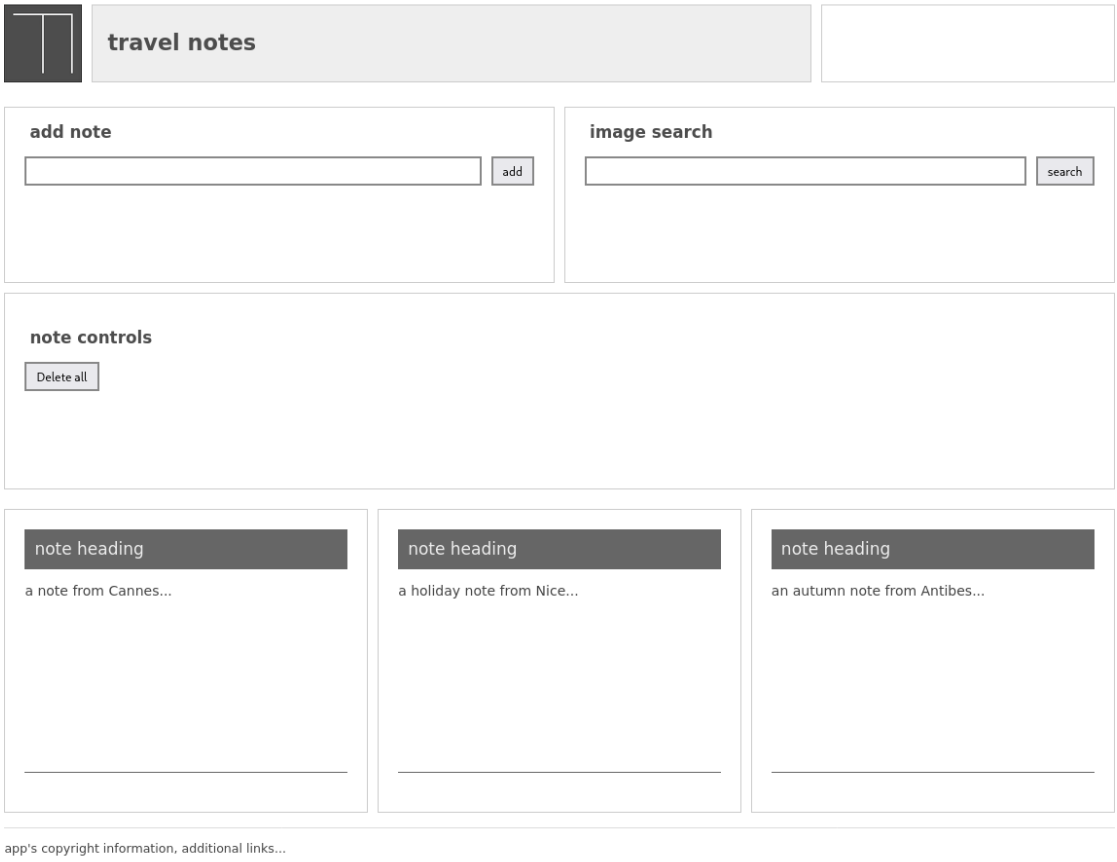

Figure 3: Travel Notes - auto-load notes

HTML5, CSS, & JS - example - part 15

add AJAX and JSON - load notes from json

- get metadata per note from JSON data

```
const loadLocalNotes = (dir, file, output) => {
    // fetch local json, promise returned
    const data = fetch(dir + file);
    // handle return promise object
    // - take response stream, read to completion
    // - call createNote for each note object
    // - note built, added to dom
    data.then(response => response.json())
        .then(jsonData => {
            console.log(jsonData);
            // get each note object
            jsonData['notes'].map((val) => {
                // GET METADATA...
                const metadata = val.metadata;
                // create note, output to dom
                createNote(val.note, output, metadata);
            });
        });
};
```

- update `createNote()` method
  - add argument for metadata per note

HTML5, CSS, & JS - example - part 16

add AJAX and JSON - load notes from json

- update `createNote()` method to handle metadata

```
function createNote(inputVal, output, metadata) {
    // define note metadata
    const date = metadata.created;
    const author = metadata.author;
    const tags = metadata.tags;

...

}
```

- then update `nodeStructure` for rendering metadata values

HTML5, CSS, & JS - example - part 17

add AJAX and JSON - load notes from json

- modify `nodeStructure` for rendering metadata values
  - node structure for tree generation

  - object = element
  - array = nested structure with first object as parent
  - nested data to `nth` depth
  - maintains hierarchy, avoids overlap &c.

```
const nodeStructure = [
    {elem: 'div', attr: 'class', attrVal: 'card-view'},
    [
        {elem: 'header', attr: 'class', attrVal: 'note-header'},
        {elem: 'h5', txtNode: 'note heading'},
    ],
    [
        {elem: 'div', attr: 'class', attrVal: 'card-content'},
        [
            {elem: 'p', txtNode: inputVal},
            [
                {elem: 'span'},
                {elem: 'a'},
            ],
        ],
    ],
    [
        {elem: 'footer'},
        {elem: 'p', attr: 'class', attrVal: 'note-metadata', txtNode: `${author} - ${date}`},
    ],
];
```

HTML5, CSS, & JS - example - part 18

add AJAX and JSON - load notes from json

- abstracted `buildNodes()` method handles update
  - does not require explicit update for metadata
- method reads `nodeStructure` and generates hierarchy
  - recursively call `arrBuilder()`
  - reduce arrays to single object per node
  - create node with value

```
...

// handles nested arrays - reduce to single obj for node creation
function arrBuilder(obj) {
    const arrNodes = obj.reduce((accumulator, currentVal) => {
        //console.log(currentVal);
        if (Array.isArray(currentVal)) {
            // recursive call - call for each nest array
            // - call until single object, then create node
            const innerArr = arrBuilder(currentVal);
            accumulator.append(innerArr);
        } else {
            const node = createNode(currentVal);
            accumulator.appendChild(node);
        }
        return accumulator;
```

```
    }, createNode(obj.shift()));
    return arrNodes;
}

...
```

- DEMO - Travel Notes - add note metadata

Image - HTML5, CSS, & JS - Travel Notes

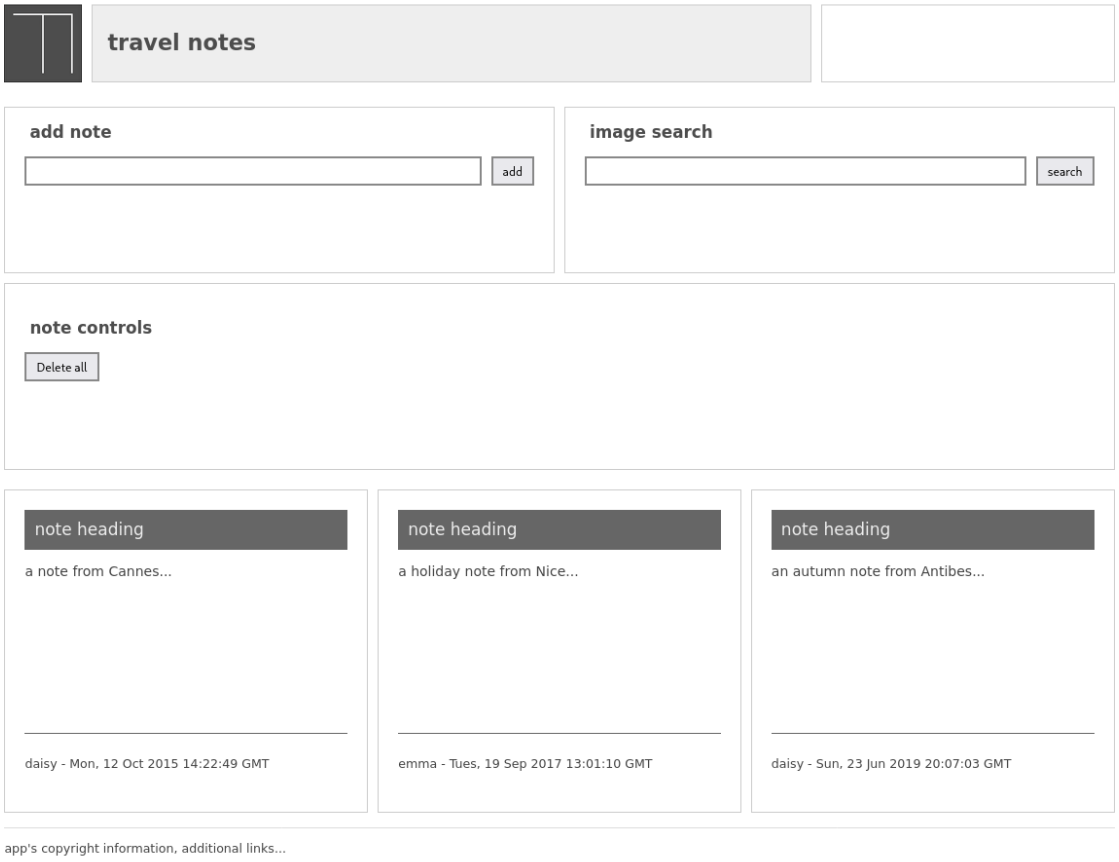

Figure 4: Travel Notes - add note metadata

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

```

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;

})
```

example - pass generator to function

- 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.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

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 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

---

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

Working with APIs - part 1

remote api options - Flickr

- **Travel Notes** app loads data from a local JSON file
- add option to load different types of data using remote APIs
  - Flickr API for images, tags…
- basics and principles are similar to the patterns we've already seen and tested
- test a sample JSON return from the Flickr API
- JSON return - useful properties for app
  - title
  - link
  - media (direct url for image - where available)
  - description
  - …
- public feed for searching public photos, videos, groups, recent activity…
- Flickr API Public Feed - Cannes and France

Working with APIs - part 2

working with Flickr API

- query Flickr's public feed for photos
  - we can use our now familiar pattern for requesting JSON
- need to pass API key to complete query

```
const flickrApiURL = 'https://www.flickr.com/services/rest/?method=flickr.photos.getRecent&api_key=' +
```

- get data from API using Fetch, handle return Promise

```
const loadFlickrURL = url => {
  // get data
    const data = fetch(url);
    // handle return Promise, use data...
    data.then(response => response.json())
            .then(jsonData => {
                // single photo for testing return data
                const photo1 = jsonData.photos.photo[0];
                // options to build photo from API
                const options = {
                    id: photo1.id,
                    secret: photo1.secret,
                    server: photo1.server,
                    size: 'm',
                };
                // create photo URL from data
                const photoURL = buildPhotoURL(options);
        // create image node, add to DOM
                const imgNode = document.createElement('img');
                const imgAttr = imgNode.setAttribute('src', photoURL);
                imageOutput.appendChild(imgNode);
            });
```

```
};

loadFlickrURL(flickrApiURL);
```

Working with APIs - part 3

working with Flickr API

- create function to build photo URL from return data

```
function buildPhotoURL(options) {
  const photoURL = 'https://live.staticflickr.com/' + options.server + '/' + options.id + '_' + options
    return photoURL;
}
```

- use same function to get different images, sizes &c.
- call function to build photo for full return data
  - e.g. iterate through data, call function to get remote image

---

Image - HTML5, CSS, & JS - Travel Notes

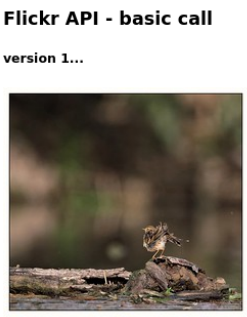

Figure 5: Flickr API - basic call for recent photos

Working with APIs - part 4

working with Flickr API

- API call may be paged for large return datasets
  - e.g. total images / photos per page
- then define page to call or default to first page
- need to render collection of images for current page

```
const loadFlickrURL = url => {
  // get data from API
    const data = fetch(url);
  // handle return Promise
    data.then(response => response.json())
            .then(jsonData => {
                // reduce data for current page
```

```
                const photos = jsonData.photos.photo.reduce((accumulator, currentVal) => {
                    // build single photo
                    const photo = buildPhoto(currentVal);
                    // append current photo to collection for page
                    accumulator.appendChild(photo);
                    // return current state of accumulator
                    return accumulator;
                }, document.createElement('div')); // define initial accumulator
                // append build photo page to DOM
                imageOutput.appendChild(photos);
            });

};
```

- update `buildPhoto()` function to create single photo

```
const buildPhoto = (data) => {
    const options = {
        id: data.id,
        secret: data.secret,
        server: data.server,
        size: 'm',
    };
    const photoURL = buildPhotoURL(options);
    const imgNode = document.createElement('img');
    const imgAttr = imgNode.setAttribute('src', photoURL);
    return imgNode;
};
```

- incrementally abstracting usage and logic

Working with APIs - part 5

working with Flickr API

- continue testing usage and abstract logic, functions &c.
- test API call for specific user
  - API key and user ID for photos
- data, config &c. passed to function
  - not relying on global variables, properties…

```
// build url for api query
const flickrApiURL = (auth, perPage, pageNo) => {
    return 'https://www.flickr.com/services/rest/?method=flickr.people.getPhotos&api_key=' + auth.key +
        '&per_page=' + perPage + '&page=' + pageNo + '&format=json&nojsoncallback=1';
}
```

- get metadata for each photo

```
// get metadata from return api data
const queryMetadata = (data) => {
    const page = data.page;
    const totalPages = data.pages;
    //const imagesPerPage = data.perpage;
    const imageTotal = data.total;
```

```
    const metaNode = document.createElement('div');
    metaNode.setAttribute('id', 'photos-metadata');
    const pageText = document.createTextNode(`page: ${page} of ${totalPages}`);
    metaNode.appendChild(pageText);
    return metaNode;
}
```

- many options for outputting metadata from API

Image - Working with APIs - Flickr API

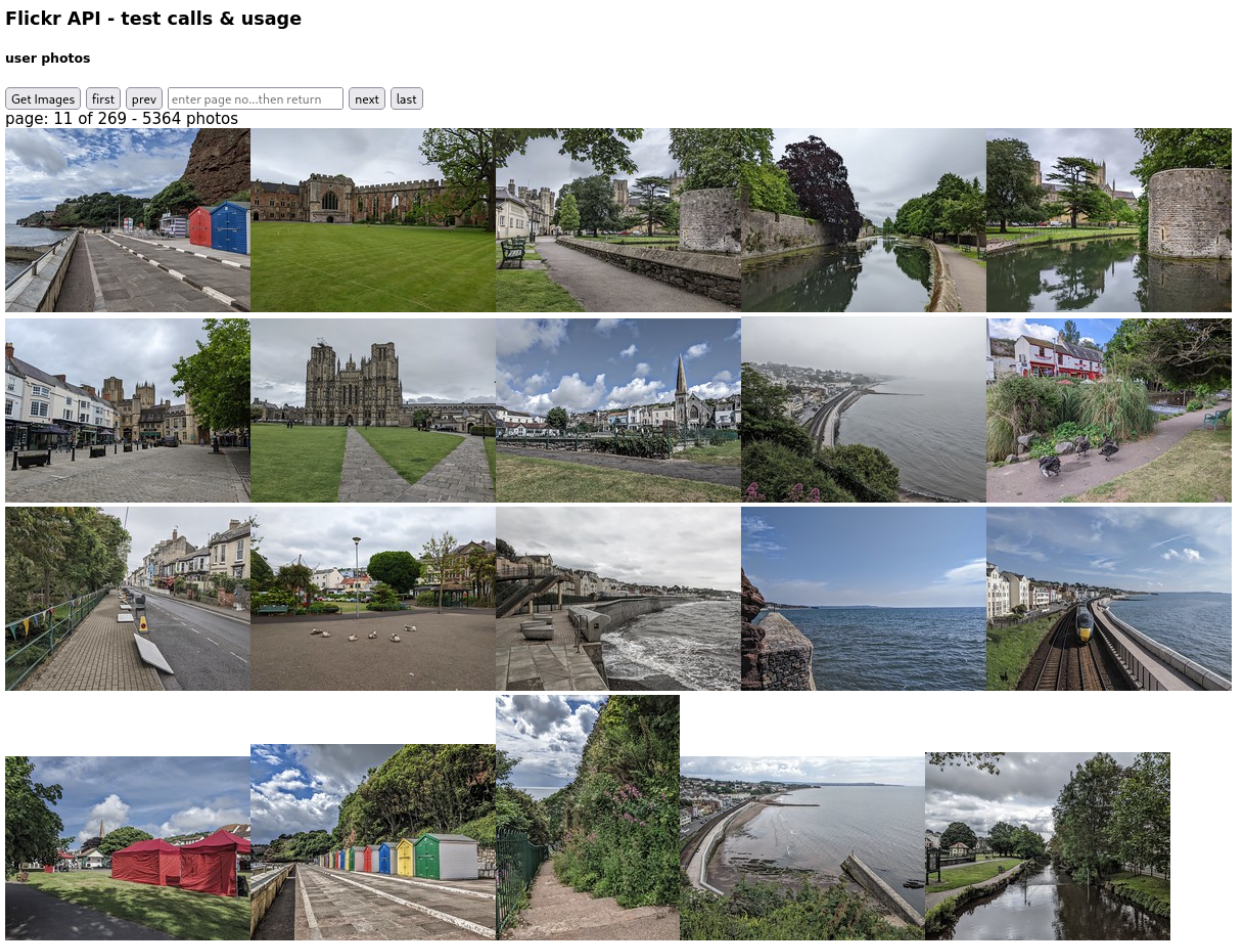

Figure 6: Flickr API - basic call for user's photos

Working with APIs - part 6

working with Flickr API

- continue moving variables, config &c. to function context
  - e.g. authOptions, page options, data &c.

- update `loadFlickrURL()` function

```
const loadFlickrURL = (pageNo)  => {
    // auth for flickr api - app garden
    // - userID to query user's photos, e.g. public photos by default
    const authOptions = {
        key: '',
        userID: '',
    };
    const perPage = 20;
    const url = flickrApiURL(authOptions, perPage, pageNo);
    const data = fetch(url);

    // use return promise data, build dom structure for image gallery
    const imgData = data.then(response => response.json())
            .then(jsonData => {
                const metadata = queryMetadata(jsonData.photos);
                const totalPages = jsonData.photos.pages;
                const photos = jsonData.photos.photo.reduce((accumulator, currentVal) => {
                    const photo = buildPhoto(currentVal);
                    accumulator.appendChild(photo);
                    return accumulator;
                }, document.createElement('div'));
                photos.setAttribute('id', 'photo-gallery');
                // check for current child nodes - remove if necessary
                while (imageOutput.firstChild) {
                    imageOutput.removeChild(imageOutput.firstChild);
                }
                imageOutput.appendChild(metadata);
                imageOutput.appendChild(photos);
                return totalPages;
            });
    return imgData;
};
```

Working with APIs - part 7

working with Flickr API

- add buttons for pager options
  - first, last, prev, next
- add option to select a page by specific number

```
// specific page no - listener for return key
pageSelector.addEventListener('keypress', (e) => {
    if (e.keyCode === 13) {
        // ADD validation for input value - type and boundaries for total nos &c....e.g. not greater tha
        pageNo = pageSelector.value;
        // basic check and validate input value - just numbers 0-9
        if (isNaN(pageNo) || pageNo > totalPages || pageNo < 1) {
            pageSelector.value= '';
            console.error('incorrect input - enter a valid page number...');
        } else {
            pageSelector.value= '';
```

```
            return loadFlickrURL(pageNo);
        }
    }
});
```

- formal validation may also be added with custom ADT for app context

---

Image - Working with APIs - Flickr API

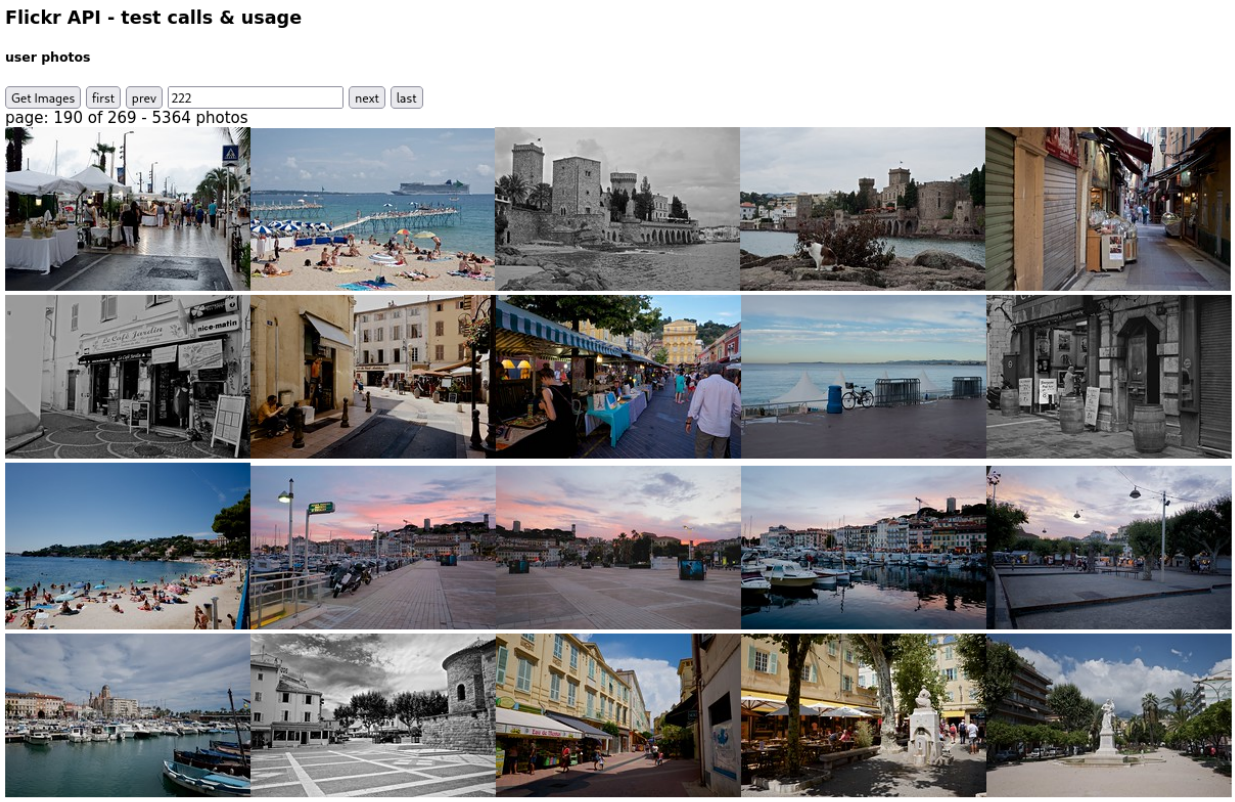

Figure 7: Flickr API - specific page from user's photos

Working with APIs - part 8

working with Flickr API

- search Flickr API by tag, text &c.
- construct URL for API query

```
const flickrApiURL = (auth, perPage, pageNo, searchQuery, sort) => {
return 'https://www.flickr.com/services/rest/?method=flickr.photos.search&api_key=' + auth.key + '&tags=
```

```
        + sort + '&safe_search=1&per_page=' + perPage + '&page=' + pageNo + '&format=json&nojsoncallback=
}
```

- `sort` parameter defined to help with first and last page of paged results
  – due to dynamic nature of returned search results...

Working with APIs - part 9

```
const searchFlickr = (pageNo, sort) => {
    // get search query from input field
    const searchQuery = searchInput.value;
    // auth for flickr api - app garden
    // - optional userID to query user's photos, e.g. public photos by default
    const authOptions = {
        key: '',
        userID: '',
    };
    // custom option can be modified to fit app requirements...
    const perPage = 20;
    // build URL for search query
    const url = flickrApiURL(authOptions, perPage, pageNo, searchQuery, sort);
    // get data from API
    const data = fetch(url);

    // use return promise data, build dom structure for image gallery
    const imgData = data.then(response => response.json())
            .then(jsonData => {
                // get metadata from return data
                const metadata = queryMetadata(jsonData.photos);
                // get total page property from return data
                const totalPages = jsonData.photos.pages;
                // reduce photos data to append to DOM
                const photos = jsonData.photos.photo.reduce((accumulator, currentVal) => {
                    // create each photo
                    const photo = buildPhoto(currentVal);
                    accumulator.appendChild(photo);
                    return accumulator;
                }, document.createElement('div')); // initial accumulator value
                // add attribute to div wrapper for created photo page
                photos.setAttribute('id', 'photo-gallery');
                // check for current child nodes - remove if necessary
                while (imageOutput.firstChild) {
                    // remove existing photos &c. - use for repeated async calls and updates
                    imageOutput.removeChild(imageOutput.firstChild);
                }
                // add metadata and images to DOM
                imageOutput.appendChild(metadata);
                imageOutput.appendChild(photos);
                // return total pages for use per async query, if necessary
                return totalPages;
            });
    return imgData;
```

```
}
```

working with Flickr API

Working with APIs - part 10

working with Flickr API

- basic `buildPhoto` method used per photo in gallery page

```
// build image for photo from return api data
const buildPhoto = (data) => {
    const options = {
        id: data.id,
        secret: data.secret,
        server: data.server,
        size: 'm',
    };
    const photoURL = buildPhotoURL(options);
    const imgNode = document.createElement('img');
    const imgAttr = imgNode.setAttribute('src', photoURL);
    return imgNode;
};
```

- initial query metadata option called for search query return data

```
// get metadata from return api data
const queryMetadata = (data) => {
    const page = data.page;
    const totalPages = data.pages;
    const imagesPerPage = data.perpage;
    const imageTotal = data.total;
    const metaNode = document.createElement('div');
    metaNode.setAttribute('id', 'photos-metadata');
    const pageText = document.createTextNode(`page: ${page} of ${totalPages}`);
    metaNode.appendChild(pageText);
    return metaNode;
}
```

- optional use of metadata properties `perpage` and `total`

Working with APIs - part 11

working with Flickr API

- call search query from button or keypress in `main` method
- define sort order for query
  - ensures newest and oldest work as expected
  - i.e. regardless of updated return search results

```
// Initialise app
const main = () => {
    // - total pages & current page no.
    let pageNo = 1;
```

```
    let totalPages = 0;
    // default sort order - newest first
    let sort = 'date-posted-desc';
    // click event - get search query
    searchImagesBtn.addEventListener('click', () => {
        // reset page no for queries after using pager buttons...
        pageNo = 1;
        const imgQuery = searchFlickr(pageNo, sort);
        imgQuery.then(pages => totalPages = pages);
    });
    // keyboard event - get search query
    searchInput.addEventListener('keypress', (e) => {
        if (e.keyCode === 13) {
            // reset page no for queries after using pager buttons...
            pageNo = 1;
            const imgQuery = searchFlickr(pageNo, sort);
            imgQuery.then(pages => totalPages = pages);
        }
    });
    firstPageBtn.addEventListener('click', () => {
        pageNo = 1;
        // reset sort order - get first page by date, newest
        sort = 'date-posted-desc';
        return searchFlickr()
    });
    prevPageBtn.addEventListener('click', () => {
        if (pageNo > 1) {
            pageNo--;
            return searchFlickr(pageNo, sort);
        }
    });
    nextPageBtn.addEventListener('click', () => {
        if (pageNo < totalPages) {
            pageNo++;
            return searchFlickr(pageNo, sort);
        }
    });
    lastPageBtn.addEventListener('click', () => {
        pageNo = 1;
        // update sort order - get last page by date, oldest
        sort = 'date-posted-asc';
        return searchFlickr(pageNo, sort);
    });
}
// load app
main();
```

Image - Working with APIs - Flickr API

Figure 8: Flickr API - specific search query

Working with APIs - part 12

working with Flickr API

- current demos
  - DEMO - version 1 - AJAX and JSON - Flickr api
  - DEMO - Flickr API - basic call - search photos

Resources

- JavaScript
  - MDN - Fetch API
  - MDN - Generators
  - MDN - Iterators and Generators
  - MDN - Promises
  - MDN - JS
- Flickr
  - Flickr API - Public feeds
  - Flickr API - Public feed - public photos & video