Comp 324/424 - Client-side Web Design

Spring Semester 2024 Week 14

Dr Nick Hayward

Final project assessment

Course total = 50 credits

- continue to develop your app concept and prototypes
- working app
  - **NO** content management systems (CMSs) such as Drupal, Joomla, WordPress…
  - **NO** PHP, Python, Ruby, C# & .Net, Java, Go, XML…
  - **NO** CSS frameworks, such as Bootstrap, Foundation, Materialize…
  - **NO** CSS preprocessors such as Sass…
  - **NO** template tools such as Handlebars.js &c.
  - must implement data from either
    - self hosted (MongoDB, Redis…)
    - APIs
    - cloud services (Firebase &c.…)
    - **NO** SQL…e.g. (you may **NOT** use MySQL, PostgreSQL &c.)
- explain design decisions
  - describe patterns used in design of UI and interaction
  - layout choices…
  - what else did you consider, and then omit? (again, why?)
- show and explain implemented differences from DEV week
  - where and why did you update the app?
  - perceived benefits of the updates?
- how did you respond to peer review?

Final project assessment

Assessment will include the following:

- final presentation & demonstration of project work
  - ~ 10 minutes per group
  - analysis of work conducted during semester
  - presentation and demonstration
    - outline state of web app concept and design
    - show final working version of web app
      - explain designs, patterns &c.
      - explain what does and does not work in the final app
      - any other pertinent information on project design & development
  - due Monday 22nd April 2024 @ 4.15pm
- final project report

- written summary of project design, development, and research
    - no word/page limit…
    - suggested report outline will be provided
    - due Monday 29th April 2024 @ 4.15pm

HTML5, CSS, & JS - example - part 12

abstract logic and components - module

- add module `flickr.js` for loading, querying &c. Flickr API
- use with defined handlers setup in `handlers.js` module
- e.g. search Flickr API for photos by tags

```
// image search
function searchFlickr(configOptions, imgInput, imgOutput, noteOutput, pageNo, sort ) {
...
}
```

- function's logic follows same usage as abstracted Flickr API example
- call custom helper functions to help structure Flickr API query
    - e.g. `flickrApiURL` , `queryMetadata` , `buildPhoto` , `createImage`
- export module for use in `handlers.js` &c.

```
export default searchFlickr;
```

- DEMO - Travel Notes - modules - search Flickr API

Image - HTML5, CSS, & JS - Travel Notes

HTML5, CSS, & JS - example - part 13

abstract logic and components - module

- additional features to consider for app design and usage
- add module `utility.js` for various extra functions and utilities
    - e.g. check exists, check visibility, hide and show, toggle, &c.
- add option to toggle between view of notes and images
- add utility functions to help check elements exist
    - use in toggle, load notes, load images &c.
- update HTML for buttons in `note-controls`

```
<!-- note controls... -->
<section class="note-controls">
    <h5>note controls</h5>
    <button id="notes-delete" class="delete-all">Delete all</button>
    <button id="notes-show" class="notes-show">Show notes</button>
    <button id="images-show" class="images-show">Show images</button>
</section>
```

- then update `utility.js` , various handlers, &c.

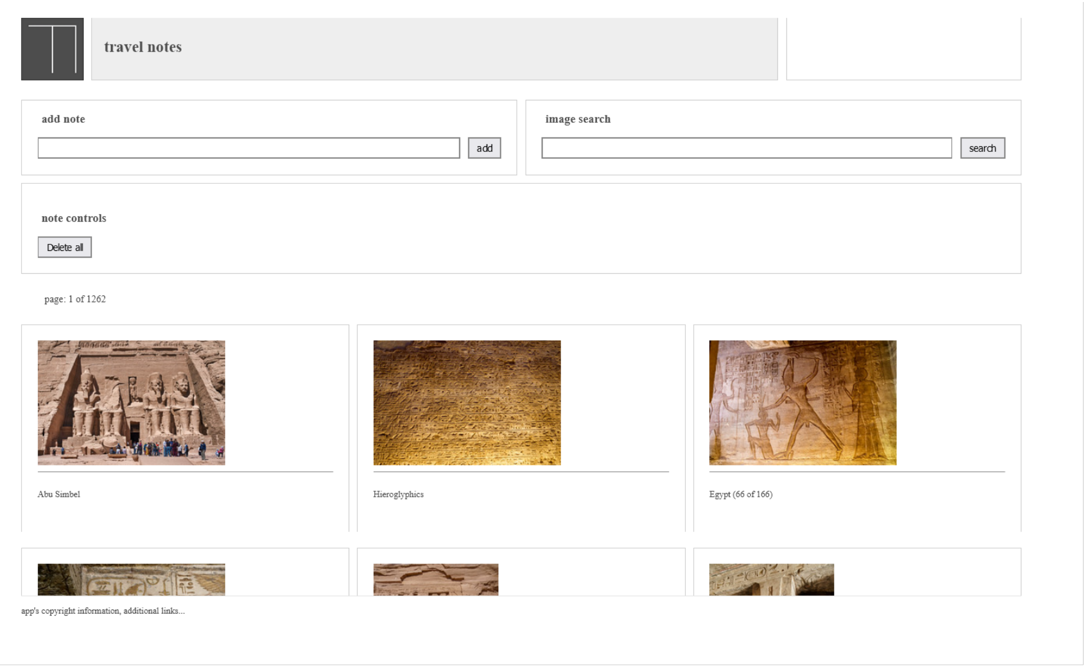


Figure 1: Travel Notes - modules - search Flickr API

HTML5, CSS, & JS - example - part 14

abstract logic and components - module

- add module for utility functions, `utility.js`
- add functions
  - `checkChildNodes` - check for nodes in passed element
  - `checkVisible` - check if passed element currently displayed in DOM
  - `elementHide` - update display of passed element to `none`
  - `elementShow` - update display of passed element to passed property, e.g. `flex`
  - `elementToggle` - toggle view between passed element to hide and passed element to show
    - option to define display property for element to show, e.g. `flex`
- currently export functions `checkChildNodes` and `elementToggle`

HTML5, CSS, & JS - example - part 15

abstract logic and components - module

- define utility functions

```
const checkChildNodes = (element) => {
    if (element.childNodes.length > 0) {
        return true;
    }
}

const checkVisible = (element) => {
```

```javascript
    // check if element displayed
    // - element shown, return true
    if (window.getComputedStyle(element).display === 'none') {
        return true;
    } else {
        return false;
    }
}

const elementHide = (element) => {
    if (checkVisible(element) === false) {
        // if shown hide element
        element.style.display = 'none';
    }
}

const elementShow = (element, option) => {
    console.log(checkVisible(element));
    if (checkVisible(element)) {
        element.style.display = option;
    }
}

const elementToggle = (elemToHide, elemToShow, option = 'flex') => {
    if (checkChildNodes(elemToHide)) {
        elementHide(elemToHide);
        elementShow(elemToShow, option);
    }
}
```

- then export required functions for use in `flickr.js` , `notes.js` , `handlers.js` &c.

```javascript
export {
    checkChildNodes,
    elementToggle
}
```

HTML5, CSS, & JS - example - part 16

abstract logic and components - module

- add toggle option to various handlers in `handlers.js`
- e.g. `imgSearchHandler`

```javascript
// Flickr API handler - image search
// args: data - api config &c.
const imgSearchHandler = (configOptions, imgInput, imgOutput, noteOutput) =>
    imgSearchBtn.addEventListener('click', () => {
    // check for empty noteOutput - no nodes
    if (checkChildNodes(noteOutput)) {
        // toggle buttons - hide images btn, show notes btn
        elementToggle(showImagesBtn, showNotesBtn, 'inline');
    }
    // get img search default from config
```

```
    searchFlickr(configOptions['flickr'], imgInput, imgOutput, noteOutput );
});
```

- and various handlers for keypress events

```
// notes handler - return keypress
const addNoteKeyHandler = (noteInput, noteOutput, imgOutput) =>
    noteInput.addEventListener('keypress', (e) => {
    if (e.keyCode === 13) {
        // check for empty imgOutput - no nodes
        if (checkChildNodes(imgOutput)) {
            // toggle buttons - hide images btn, show notes btn
            elementToggle(showNotesBtn, showImagesBtn, 'inline');
        }
        newNote(noteInput, noteOutput, imgOutput);
    }
});
```

HTML5, CSS, & JS - example - part 17

abstract logic and components - module

- add handler functions for buttons to show notes and show images
  - `showNotesBtnHandler` and `showImagesBtnHandler`

```
// notes handler - show notes on button click
const showNotesBtnHandler = (imgOutput, noteOutput) =>
    showNotesBtn.addEventListener('click', () => {
    // toggle content output - show notes, hide images
    elementToggle(imgOutput, noteOutput);
    // toggle buttons - show images, hide notes
    elementToggle(showNotesBtn, showImagesBtn, 'inline');
});
// images handler - show images on button click
const showImagesBtnHandler = (imgOutput, noteOutput) =>
    showImagesBtn.addEventListener('click', () => {
    elementToggle(noteOutput, imgOutput);
    elementToggle(showImagesBtn, showNotesBtn, 'inline');
});
```

- checks for current visible nodes &c. part of logic for `elementToggle` function

```
const checkChildNodes = (element) => {
    if (element.childNodes.length > 0) {
        return true;
    }
}
const checkVisible = (element) => {
    // check if element displayed
    // - element shown, return true
    if (window.getComputedStyle(element).display === 'none') {
        return true;
    } else {
        return false;
    }
```

```js
}
const elementHide = (element) => {
    if (checkVisible(element) === false) {
        // if shown hide element
        element.style.display = 'none';
    }
}
const elementShow = (element, option) => {
    console.log(checkVisible(element));
    if (checkVisible(element)) {
        element.style.display = option;
    }
}
const elementToggle = (elemToHide, elemToShow, option = 'flex') => {
    if (checkChildNodes(elemToHide)) {
        elementHide(elemToHide);
        elementShow(elemToShow, option);
    }
}
```

- DEMO - Travel Notes - modules - toggle view option

Image - HTML5, CSS, & JS - Travel Notes

Image - HTML5, CSS, & JS - Travel Notes

HTML5, CSS, & JS - example - part 18

abstract logic and components - module

- update UI for fixed search options
- update main content scroll
- persist 'add note', 'image search', and 'note controls' in UI
  - allow main content to scroll beneath header and page heading
- user may easily scroll content
  - then search for different images
  - add more notes
  - quickly switch between images and notes
- ...

HTML5, CSS, & JS - example - part 19

abstract logic and components - module

- move `page-heading` out of `main` content

```html
<body>
    <!-- grid wrapper -->
    <div class="wrapper">
```

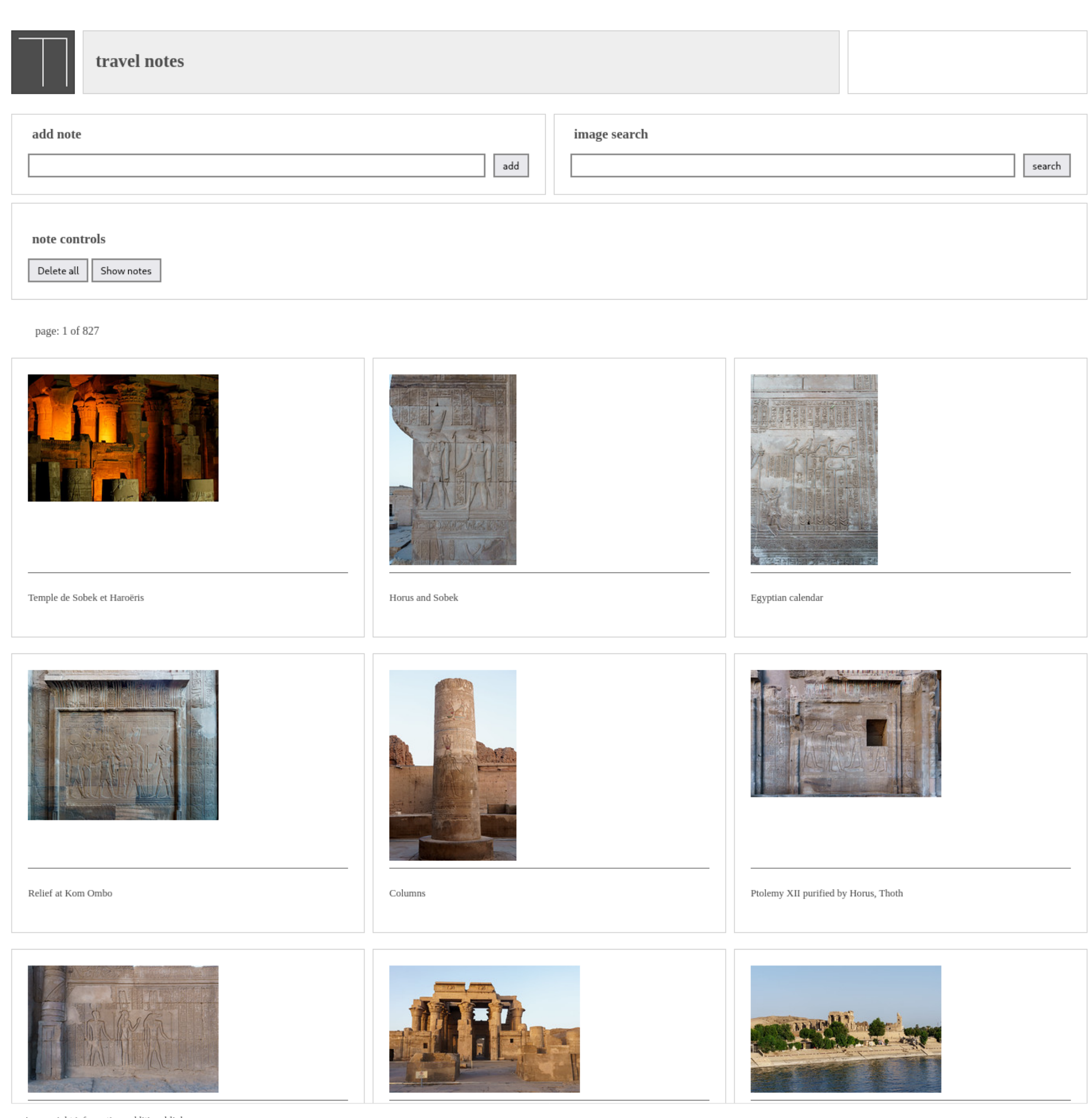


Figure 2: Travel Notes - modules - search Flickr API - toggle notes

Figure 3: Travel Notes - modules - search Flickr API - toggle images

```
        <!-- grid banner -->
            <!-- logo -->
            <!-- document header -->
            <!-- banner extras -->
        <div class="page-heading">
            <!-- note input -->
            <section class="note-input">
                <h5>add note</h5>
                <div class="input-group">
                <input type="text" id="input-note" />
                <button id="add-note">add</button>
                </div>
            </section>
            <!-- image search -->
            <section class="image-search">
                <h5>image search</h5>
                <div class="input-group">
                <input type="text" id="input-image" />
                <button id="search-images">search</button>
                </div>
            </section>
            <!-- note controls for delete, view toggles... -->
            <section class="note-controls">
            <h5>note controls</h5>
            <button id="notes-delete" class="delete-all">Delete all</button>
            <button id="notes-show" class="notes-show">Show notes</button>
            <button id="images-show" class="images-show">Show images</button>
            </section>
        </div><!-- end of page-heading -->
    <!-- document main - unique to current page -->
        <!-- document footer -->
    </div> <!-- end of grid wrapper -->
    <!-- js scripts... -->
   <script type="module" src="./index.js"></script>
</body>
```

HTML5, CSS, & JS - example - part 19

abstract logic and components - module

- update CSS for grid layout
  - `wrapper`
  - add extra template row value for `page-heading`
  - add extra template area for `page-heading`
  - `page-heading` moved above `site-content`

```
div.wrapper {
    ...
    grid-template-rows: 80px 160px auto 80px;
    grid-template-areas:
        "site-banner"
        "page-heading"
```

```
        "site-content"
        "footer";
    ...
}
```

- `page-heading`
    - group template areas for `add-note search images` and `note-controls`

```
div.page-heading {
    ...
    grid-template-columns: 50% 50%;
    grid-template-areas:
        "add-note search-images"
        "note-controls note-controls";
    ...
}
```

- `site-content`
    - remove previous `page-heading` and limit to `content`

```
.site-content {
    ...
    grid-template-areas:
        "content";
}
```

HTML5, CSS, & JS - example - part 20

abstract logic and components - module

- update responsive CSS for layout `915px`
    - add template row for `page-heading` before content
    - add `160px` for `page-heading` grouping

```
div.wrapper {
    grid-template-rows: 80px 160px auto 80px;
    margin: 20px 10px 0 10px;
}

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

- update responsive CSS for layout `545px`
    - add template row for `page-heading`
    - add `265px` for three rows in `page-heading`

```
div.wrapper {
    grid-template-rows: 80px 265px auto;
    margin: 5px 0 0 0;
```

```
}
```

- update `page-heading` for three `100%` rows
  - stacked rows
  - fixed height for group

```
div.page-heading {
    grid-template-columns: 100%;
    grid-template-areas:
        "add-note"
        "search-images"
        "note-controls";
    height: 265px;
}
```

- DEMO - Travel Notes - modules - update design

Image - HTML5, CSS, & JS - Travel Notes

Figure 4: Travel Notes - modules - update UI - desktop layout

---

Image - HTML5, CSS, & JS - Travel Notes

Image - HTML5, CSS, & JS - Travel Notes

Systems Management - Build Tools & Project Development

Extra notes

- Systems
  - Environments & Distributions
  - Build first - overview and usage
- Grunt
  - basics
  - integrate with project outline and development
  - integrate with project release
- Webpack
  - setup for local project
  - basic usage
  - assets for local project
  - …

JavaScript - Prototype

intro

- along with the following traits of JS (ES6 …),
  - functions as first-class *objects*
  - versatile and useful structure of functions with closures
  - combine generator functions with promises to help manage async code
  - async & await…
- *prototype* object may be used to delegate the search for a particular property
- a *prototype* is a useful and convenient option for defining properties and functionality
  - accessible to other objects
- a *prototype* is a useful option for replicating many concepts in traditional object oriented programming

---

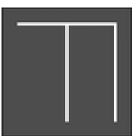
JavaScript - Prototype

understanding prototypes

- in JS, we may create objects, e.g. using *object-literal* notation
  - a simple value for the first property
  - a function assigned to the second property
  - another object assigned to the third object

```
let testObject = {
    property1: 1,
    prooerty2: function() {},
    property3: {}
}
```

- as a dynamic language, JS will also allow us to



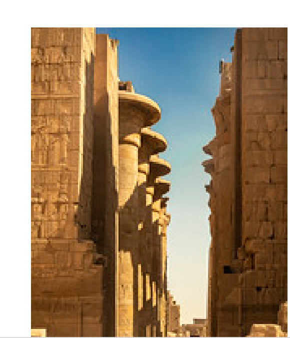


Figure 5: Travel Notes - modules - update UI - tablet layout

travel notes

add note

add

image search

search

note controls

Delete all Show notes

Book of Caverns (Detail)

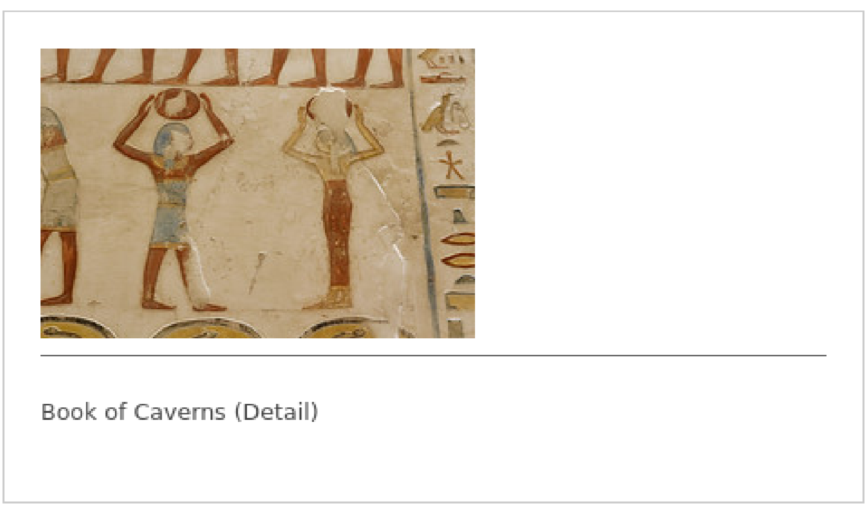


Figure 6: Travel Notes - modules - update UI - tablet layout

    - modify these properties
    - delete any not required
    - or simply add a new one as necessary
- this dynamic nature may also completely change the properties in a given object
- this issue is often solved in traditional object-oriented languages using inheritance
- in JS, we can use *prototype* to implement inheritance

JavaScript - Prototype

basic idea of prototypes

- every *object* can have a reference to its *prototype*
    - a delegate object with properties - default for child objects
- JS will initially search the onject for a property
    - then, search the *prototype*
    - i.e. prototype is a fall back object to search for a given property &c.

```
const object1 = { title: 'the glass bead game' };
const object2 = { author: 'herman hesse' };

console.log(object1.title);

Object.setPrototypeOf(object1, object2);

console.log(object1.author);
```

- in the above example, we define two objects
    - properties may be called with standard object notation
    - can be modified and mutated as standard
    - use `setPrototypeOf()` to set and update object's prototype
- e.g. `object1` as object to update
    - `object2` as the object to set as prototype
- if requested property is not available on `object1`
    - JS will search defined prototype…
- `author` available as property of prototype for `object1`
- demo - basic prototype

JavaScript - Prototype

prototype inheritance

- *Prototypes*, and their properties, can also be inherited
    - creates a chain of inheritance…
- e.g.

```
const object1 = { title: 'the glass bead game' };
const object2 = { author: 'herman hesse' };
const object3 = { genre: 'fiction' };

console.log(object1.title);

Object.setPrototypeOf(object1, object2);
Object.setPrototypeOf(object2, object3);
```

```
console.log(object1.author);
console.log(`genre from prototype chain = ${object1.genre}`); // use template literal to output...
```

- `object1` has access to the prototype of its parent, `object2`
- a property search against `object1` will now include its own prototype, `object2`
  - and its prototype as well, `object3`
- output for `object1.genre` will return the value stored in the property on `object3`
- demo - basic set prototype

JavaScript - Prototype

object constructor & prototypes

- object-oriented languages, such as Java and C++, include a class constructor
  - provides known encapsulation and structuring
  - constructor is initialising an object to a known initial state...
- i.e. consolidate a set of properties and methods for a class of objects in one place
- JS offers such a mechanism, although in a slightly different form to Java, C++ &c.
- JS still uses the `new` operator to instantiate new objects via constructors
  - JS does not include a true class definition comparable to Java &c.
  - ES6 `class` is syntactic sugar for the `prototype` ...
- `new` operator in JS is applied to a constructor function
  - this triggers the creation of a new object

JavaScript - Prototype

prototype object

- in JS, every function includes their own prototype object
  - set automatically as the prototype of any created objects
  - e.g.

```
//constructor for object
function LibraryRecord() {
  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

const bookRecord = new LibraryRecord();

console.log(bookRecord.library);
```

- likewise, we may set a default method on an instantiated object's prototype
- demo - basic prototype object

JavaScript - Prototype

instance properties

- as JS searches an object for properties, values or methods
  - instance properties will be searched before trying the prototype

    - a known order of precedence will work.
    - e.g.

```
//constructor for object
function LibraryRecord() {
  // set property on instance of object
  this.library = 'waldzell';

  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

const bookRecord = new LibraryRecord();

console.log(bookRecord.library);
```

- `this` refers directly to the newly created object
    - properties in constructor created directly on instantiated object
    - e.g. instance of `LibraryRecord()`
- search for `library` property against object
    - do not need to search against prototype for this example
- known side-effect
    - instantiate multiple objects with this constructor
    - each object gets its own copy of the constructor's properties & access to same prototype
    - may end up with multiple copies of same properties in memory
- if replication is required or likely
    - more efficient to store properties & methods against the prototype
- demo - basic prototype object properties

JavaScript - Prototype

side effects of JS dynamic nature

- JS is a dynamic language
    - properties can be added, removed, modified…
- dynamic nature is true for prototypes
    - function prototypes
    - object prototypes

```
//constructor for object
function LibraryRecord() {
  // set property on instance of object
  this.library = 'waldzell';
}

// create instance of LibraryRecord - call constructor with `new` operator
const bookRecord1 = new LibraryRecord();

// check output of value for library property from constructor
console.log(`this library = ${bookRecord1.library}`);

// add method to prototype after object created
LibraryRecord.prototype.updateLibrary = function() {
  return this.retreat = 'mariafels';
```

```
};

// check prototype updated with new method
console.log(`this retreat = ${bookRecord1.updateLibrary()}`);

// then overwrite prototype - constructor for existing object unaffected...
LibraryRecord.prototype = {
  archive: 'mariafels',
  order: 'benedictine'
};

// create instance object of LibraryRecord...with updated prototype
const bookRecord2 = new LibraryRecord();

// check output for second instance object
console.log(`updated archive = ${bookRecord2.archive} and order = ${bookRecord2.order}`);
// check output for second instance object - library
console.log(`second instance object - library = ${bookRecord2.library}`);
// check if prototype updated for first instance object - NO
console.log(`first instance object = ${bookRecord1.order}`);
// manual update to prototype for first instance object still available
console.log(`this retreat2 = ${bookRecord1.updateLibrary()}`);

// check prototype has been fully overwritten
// - e.g. `updateLibrary()` no longer available on prototype for new instance object
try {
// updates to original prototype are overridden
// - error is returned for second instantiated object...
console.log(`this retreat = ${bookRecord2.updateLibrary()}`);
 } catch(error) {
  console.log(`modified prototype not available for new object...\n ${error}`);
 }
```

- demo - basic prototype dynamic

JavaScript - Prototype

object typing via constructors

- check function used as a constructor to instantiate an object
  - using `constructor` property

```
//constructor for object
function LibraryRecord() {
  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

// create instance object for libraryRecord
const bookRecord = new LibraryRecord();

// output constructor for instance object
console.log(`constructor = ${bookRecord.constructor}`);
```

```
// check if function was constructor (use ternary conditional)
const check = bookRecord.constructor === LibraryRecord ? true : false;
// output result of check
console.log(check);
```

- demo - basic constructor check

JavaScript - Prototype

instantiate a new object using a constructor reference

- use a constructor to create a new instance object
- also use `constructor()` of new object to create another object
- second object is still an object of the original constructor

```
//constructor for object
function LibraryRecord() {
  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

const bookRecord = new LibraryRecord();
const bookRecord2 = new bookRecord.constructor();
```

JavaScript - Prototype

achieving inheritance

- *Inheritance* enables re-use of an object's properties by another object
- helps us efficiently avoid repetition of code and logic
  - improving reuse and data across an application
- in JS, a prototype chain to ensure inheritance works beyond simply copying prototype properties
  - e.g. a book in a corpus, a corpus in an archive, an archive in a library…

JavaScript - Prototype

inheritance with prototypes - part 1

- *inheritance* in JS
  - create a prototype chain using an instance of an object as prototype for another object
  - e.g.

```
SubClass.prototype = new SuperClass()
```

- this pattern works as a prototype chain for inheritance
  - prototype of `SubClass` instance as an instance of `SuperClass`
  - prototype will have all the properties of `SuperClass`
  - `SuperClass` may also have properties from its superclass…
- prototype chain created of expected inheritance

JavaScript - Prototype

inheritance with prototypes - part 2

- e.g. inheritance achieved by setting prototype of `Archive` to instance of `Library` object

```
//constructor for object
function Library() {
    // instance properties
  this.type = 'library';
  this.location = 'waldzell';
}

// constructor for Archive object
function Archive(){
    // instance property
  this.domain = 'gaming';
}

// update prototype to parent Libary - instance relative to parent & child
Archive.prototype = new Library();

// instantiate new Archive object
const archiveRecord = new Archive();

// check instance object - against constructor
if (archiveRecord instanceof Archive) {
  console.log(`archive domain = ${archiveRecord.domain}`);
}

// check instance of archiveRecord - instance of Library & Archive
if (archiveRecord instanceof Library) {
    // type property from Library
  console.log(`Library type = ${archiveRecord.type}`);
    // domain property from Archive
    console.log(`Archive domain = ${archiveRecord.domain}`);
}
```

JavaScript - Prototype

issues with overriding the constructor property

- setting `Library` object as defined prototype for `Archive` constructor

```
Archive.prototype = new Library();
```

- connection to `Archive` constructor **lost** - we may check constructor

```
// check constructor used for archiveRecord object
if (archiveRecord.constructor === Archive) {
  console.log('constructor found on Archive...');
} else {
    // Library constructor output - due to prototype
  console.log(`Archive constructor = ${archiveRecord.constructor}`);
}
```

- `Library` constructor will be returned
  - *n.b.* may become an issue - constructor property may be used to check original function for instantiation
- demo - inheritance with prototype

JavaScript - Prototype

```
//constructor for object
function Library() {
    // instance properties
  this.type = 'library';
  this.location = 'waldzell';
}

// extend prototype
Library.prototype.addArchive = function(archive) {
  console.log(`archive added to library - ${archive}`);
    // add archive property to instantiate object
    this.archive = archive;
    // add property to Library prototype
    Library.prototype.administrator = 'knechts';
}

// constructor for Archive object
function Archive(){
    // instance property
  this.domain = 'gaming';
}

// update prototype to parent Libary - instance relative to parent & child
Archive.prototype = new Library();

// instantiate new Archive object
const archiveRecord = new Archive();
// call addArchive on Library prototype
archiveRecord.addArchive('mariafels');

// check instance object - against constructor
if (archiveRecord instanceof Archive) {
  console.log(`archive domain = ${archiveRecord.domain}`);
}

// check constructor used for archiveRecord object
if (archiveRecord.constructor === Archive) {
  console.log('constructor found on Archive...');
} else {
  console.log(`Archive constructor = ${archiveRecord.constructor}`);
    console.log(`Archive domain = ${archiveRecord.domain}`);
    console.log(`Archive = ${archiveRecord.archive}`);
    console.log(`Archive admin = ${archiveRecord.administrator}`);
}
```

```js
// check instance of archiveRecord - instance of Library & Archive
if (archiveRecord instanceof Library) {
    // type property from Library
  console.log(`Library type = ${archiveRecord.type}`);
    // domain property from Archive
    console.log(`Archive domain = ${archiveRecord.domain}`);
}

// instantiate another Archive object
const archiveRecord2 = new Archive();
// output instance object for second archive
console.log('Archive2 object = ', archiveRecord2);
// check if archiveRecord2 object has access to updated archive property...NO
console.log(`Archive2 = ${archiveRecord2.archive}`);
// check if archiveRecord2 object has access to updated adminstrator property...YES
console.log(`Archive2 administrator = ${archiveRecord2.administrator}`);
```

some benefits of overriding the constructor property

- demo - inheritance with prototype - updated

JavaScript - Prototype

configure object properties - part 1

- each object property in JS is described with a **property descriptor**
- use such descriptors to configure specific keys, e.g.
- *configurable* - boolean setting
  - true = property's descriptor may be changed and the property deleted
  - false = no changes &c.
- *enumerable* - boolean setting
  - true = specified property will be visible in a `for-in` loop through object's properties
- *value* - specifies value for property (default is undefined)
- *writable* - boolean setting
  - true = the property value may be changed using an assignment
- *get* - defines the getter function, called when we access the property
  - **n.b.** can't be defined with *value* and *writable*
- *set* - defines the setter function, used whenever an assignment is made to the property
  - **n.b.** can't be defined with *value* and *writable*
- e.g. create following property for an object

```js
archive.type = 'private';
```

- `archive`
  - will be *configurable*, *enumerable*, *writable*
  - with a value of *private*
  - *get* and *set* will currently be undefined

JavaScript - Prototype

configure object properties - part 2

- to update or modify a property configuration use built-in `Object.defineProperty()` method

- this method takes an object, which may be used to
  - define or update the property
  - define or update the name of the property
  - define a property descriptor object
  - e.g.

```
// empty object
const archive = {};

// add properties to object
archive.name = "waldzell";
archive.type = "game";

// define property access, usage, &c.
Object.defineProperty(archive, "access", {
    configurable: false,
    enumerable: false,
    value: true,
    writable: true
});

// check access to new property
console.log(`${archive.access}, access property available on the object...`);

/*
* check we can't access new property in loop
* - for..in iterates over enumerable properties
*/
for (let property in archive) {
    // log enumerable
    console.log(`key = ${property}, value = ${archive[property]}`);
}

/*
* plain object values not iterable...
* - returns expected TyoeError - archive is not iterable
*/
for (let value of archive) {
    // value not logged...
    console.log(value);
}
```

- demo - configure object properties

HTML5, CSS, & JS - example - part 1

use Prototype for Validation

- update module structure to add Validation
- add `validation.js` to `src` directory
- add `import` for Validation
  - e.g. `handlers.js`

```
import { checkObjType } from './validation.js';
```

HTML5, CSS, & JS - example - part 2

use Prototype for Validation

- define parent `Validation` in `validation.js`

```
/* Validation object
*/
function Validation(value) { // validation constructor
    this.val = value;
}
// Validation - of
// - lifting operation for success...
Validation.of = function(value) { return Success.of(value) }

// Validation - get
// - inherited by Success & Failure objects
Validation.prototype.get = function() { return this.val }

// Validation - success
// - create Success object
Validation.prototype.success = function(a) { return Success.of(a) }

// Validation - failure
// - create Failure object
Validation.prototype.failure = function(error) { return Failure.of(error) }

// Validation - isSuccess
// - get Success
// - default return
// - override in Success object
Validation.prototype.isSuccess = false;

// Validation - isFailure
// - get Failure
// - default return
// - override in Failure object, where appropriate
Validation.prototype.isFailure = false;
```

HTML5, CSS, & JS - example - part 3

use Prototype for Validation

- define logic for child objects
  - Success

```
// Success - constructor
function Success(value) {
    this.val = value;
}
```

```
// Success - of
// - lifting operation
// - instantiate Success object
Success.of = function(value) { return new Success(value) }
// Success - isSuccess
// - boolean overrides default set by Validation
Success.prototype.isSuccess = true;
```

- Failure

```
// Failure - constructor
// - value set as custom error message, error code &c.
function Failure(value) {
    this.val = value;
}
// Failure - of
// - lifting operation
// - instantiate custom Failure object
Failure.of = function(value) { return new Failure(value) }
// Failure - isFailure
// - boolean overrides default set by Validation
Failure.prototype.isFailure = true;
```

HTML5, CSS, & JS - example - part 4

use Prototype for Validation

- set Prototype of Success to parent

```
// Success - update Prototype
// - set [[Prototype]] of Success.prototype to Validation.prototype
// - similar usage to `Class extends`
Object.setPrototypeOf(Success.prototype, Validation.prototype);
```

- set Prototype of Failure to parent

```
// Failure - update Prototype
// - set prototype to parent Validation
Object.setPrototypeOf(Failure.prototype, Validation.prototype);
```

HTML5, CSS, & JS - example - part 5

use Prototype for Validation

- add test for usage of Validation
- define basic check of passed object

```
// Check passed object - validate for type &c.
const checkObjType = (obj) => {
    const data = Object.values(obj);
    console.log(data);
    const check = data.reduce((accumulator, currentVal) => {
//      if (typeof currentVal !== 'string') {
    // basic test - check logic works as expected...
```

```
    // - string will return true for accumulator
    if (isNaN(currentVal)) {
            accumulator = true;
        }
        return accumulator;
    }, false ); // default accumulator set to false for numbers...
    return check === true
    ? Success.of(obj)
    : Failure.of('type did not match for properties...');
}
```

- check can include any custom requirement
  - e.g. specific to types, data store schema, &c.
- return for `check` instantiates object wrapper for value
  - e.g. instantiate `Success` object for success
  - instantiate `Failure` object for failure in check

HTML5, CSS, & JS - example - part 6

use Prototype for Validation

- export from `validation.js`
  - in this context only need to export function to check

```
export {
    checkObjType
};
```

- good example of Module usage
  - closures, bindings, exports...

HTML5, CSS, & JS - example - part 7

use Prototype for Validation

- use check function `checkObjType` in `handlers.js`
- e.g. update handler for add note button

```
// notes handler - add button
const addNoteBtnHandler = (noteInput, noteOutput, imgOutput)
    => addNoteBtn.addEventListener('click', () => {
    // validate input data - wrap data in plain object for validation use...
    const validateObj = checkObjType({
        value: noteInput.value});
    // check validation return - Success or Failure obj...
    if (validateObj.isSuccess) {
        console.log(validateObj.get());
        // check for empty imgOutput - no nodes
        if (checkChildNodes(imgOutput)) {
            // toggle buttons - hide images btn, show notes btn
            elementToggle(showNotesBtn, showImagesBtn, 'inline');
        }
        // pass Success obj to create new note
```

```
        newNote(validateObj, noteOutput, imgOutput);
    } else {
        // log return Failure obj for validation
        console.log(validateObj.get());
    }
    noteInput.value = "";
});
```

HTML5, CSS, & JS - example - part 8

use Prototype for Validation

- use imported `checkObjType()` to check and return Validation object
  - either Success or Failure object
- then check success for returned Validation object

```
...
// check validation return - Success or Failure obj...
if (validateObj.isSuccess) {
...
}
...
```

- property `isSuccess` available on parent `Validation`
  - default property value updated by Success
- then pass successful validation object to functions

```
// pass Success obj to create new note
newNote(validateObj, noteOutput, imgOutput);
```

- `validateObj` is wrapped as Success object
  - has access to properties and methods
  - defined in `validation.js` for parent Validation

HTML5, CSS, & JS - example - part 9

use Prototype for Validation

- use Validation object for Success return in functions with data
  - e.g. `newNote()`

```
const newNote = (noteInput, noteOutput, imgOutput) => {
    const date = new Date().toUTCString();
    const metadata = {
        "created": date,
        "author": "daisy",
        /*"tags": "note"*/
    };
    // get value from Success obj
  // - forces use of validation obj and not direct execution with value...
    const data = noteInput.get();
    createNote(data.value, noteOutput, metadata, imgOutput);
};
```

- passing Validation object forces use of method `get()`

- used to access `value` property wrapped in Success
- prevents direct use of method to pass unvalidated data
- DEMO - Travel Notes - modules - add validation

Image - HTML5, CSS, & JS - Travel Notes

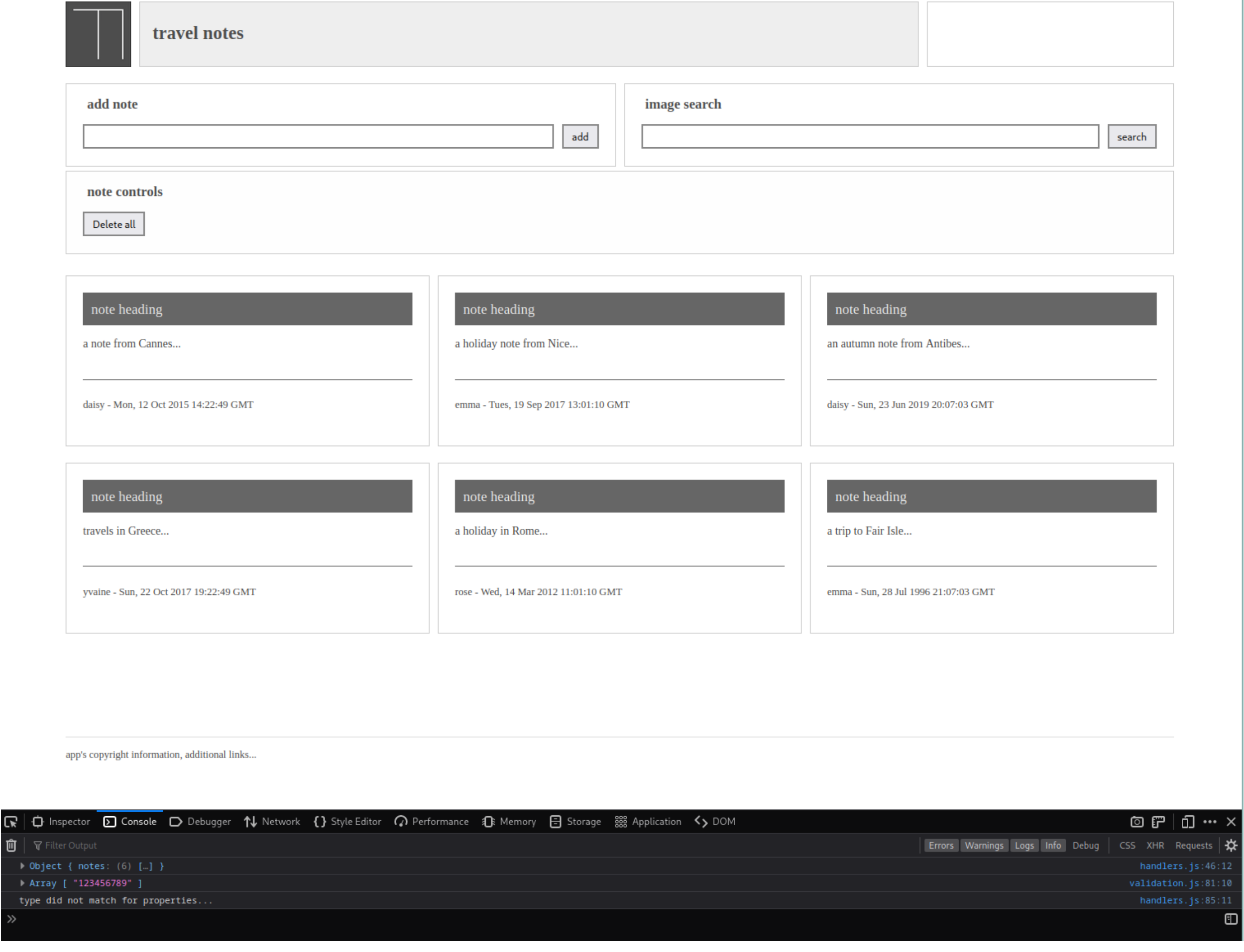


Figure 7: Travel Notes - modules - add validation

HTML5, CSS, & JS - example - part 10

use Prototype for Validation

- Validation may now be used throughout app
  - e.g. check any data inputs, datastore I/O &c.
- add validation to current app
- `/src/app.js`
  - validation for overall app loading and execution
  - bootstrap app from `config.js` & `config.json`
  - use return validation from `config.js`
  - `isSuccess` - use data

- `isFailure` - return error and exit app load
- `/src/flickr.js`
  - part 1 - check and validate input data for image search, searchQuery variable
  - return wrapped validation obj Success / Failure
  - check `isSuccess` / `isFailure`
  - add handlers for return validation obj
  - part 2 - check and validate return json data from fetch
  - return Success / Failure obj for data
  - check `isSuccess` / `isFailure`
- `src/handlers.js`
  - `loadLocalJSON()` and `createNote()`

JavaScript - Prototype

using ES Classes

- ES6 provides a new `class` keyword
  - enables object creation and aida in inheritance
  - it's *syntactic sugar* for the prototype and instantiation of objects
  - e.g.

```
// class with constructor & methods
class Archive {
  constructor(name, admin) {
    this.name = name;
      this.admin = admin;
  }
    // class method
  static access() {
    return false;
  }
    // instance method
    administrator() {
        return this.admin;
    }
}

// instantiate archive object
const archive = new Archive('Waldzell', 'Knechts');

// check parameter usage with class
const nameCheck = archive.name === `Waldzell` ? archive.name : false;

// log archive name
console.log(`class archive name = ${nameCheck}`);
// call class method
console.log(Archive.access());
// call instance method
console.log(`archive administrator = ${archive.administrator()}`);
```

- demo - basic ES Class

JavaScript - Prototype

ES classes as syntactic sugar

- classes in ES6 are simply syntactic sugar for prototypes.
- a prototype implementation of previous `Archive` class, and usage… -not* e.g.

```
// constructor function
function Archive(name, admin) {
  this.name = name;
    this.admin = admin;

    // instance method
    this.administrator = function () {
        return this.admin;
    }

    // add property to constructor
    Archive.access = function() {
    return false;
    };
}

// instantiate object - pass arguments
const archive = new Archive('Waldzell', 'Knechts');

// check parameter usage with ternary conditional...
const nameCheck = archive.name === `Waldzell` ? archive.name : false;

// output name check...
console.log(`prototype archive name = ${nameCheck}`);
// call constructor only method
console.log(Archive.access());
// call instance method
console.log(`archive administrator = ${archive.administrator()}`);
```

- demo - basic Prototype equivalent

JavaScript - Proxy

intro

- use a *proxy* to control access to another object
  - a surrogate relationship between the proxy and the object
- proxy may be considered akin to a generalised *getter* and *setter*
- whilst *getters* and *setters* may control access to a single object property
  - a proxy enables generic handling of interactions
- interactions may even include method calls relative to an object
- we may use a proxy where we might otherwise use a getter and a setter
- proxy is considered broader and more powerful in its potential implementation and usage
- e.g.
  - a proxy may be used to add profiling support to an object
  - measure performance
  - autopopulate code properties
  - …

JavaScript - Proxy

creating a proxy - part 1

- to create a proxy in JavaScript
  - use the default, built-in Proxy constructor

```
// plain object
const planet = {
  name: ['mercury'],
  codes: {
    iau: 'Me',
    unicode: 'U+263F'
  }
};

// proxy for passed target object - target = planet
const planetDetails = new Proxy(planet, {
  get: (target, key) => {
    return key in target ? target[key] :'planet does not exist...';
  },
  set: (target, key, value) => {
    key in target ? target[key].push(value) : 'key not found...';
  }
});

// check proxy access to target property
console.log(planetDetails.name);

// check proxy set against target property
// target = planet, key = name, value = earth
planetDetails.name = 'earth';

console.log(planetDetails.name);
```

JavaScript - Proxy

creating a proxy - part 2

- in the previous example
  - we may access the object and its properties directly
  - but the proxy gives us extra utility
- e.g for the getter and setter
  - we may check keys, values, &c.
  - control how the object is updated
  - we may also add basic logging, if necessary…
- after defining the initial plain object, `planet`
  - we may then wrap it using the Proxy constructor
- current proxy includes a getter and setter method
  - contains checks for required key in the original object
- also choose how we would like to compute values, log usage and return &c.

JavaScript - Proxy

proxy traps

- in the previous example
  - we added a `get` and `set` trap for defined target object, `planet`
- there are other traps we may use with a Proxy
- e.g.
  - `apply` - activated for a function call
    - e.g. measuring performance
  - `construct` - activated for `new` keyword
  - `enumerate` - activated for `for-in` statements
  - `getPrototypeOf` - activated for getting prototype value
  - `setPrototypeOf` - activated for setting prototype value
- these traps are in addition to existing `get` and `set` traps
- there are also traps that we cannot override using a proxy
- e.g.
  - equality operators - `==` and `===` and not equivalents
  - `instanceof` and `typeof`

JavaScript - Proxy

logging with proxies

- use logging in development as a convenient tool for debugging and checking code
- output checks, and add debugging statements to various points within our code
- quickly start to add many such logging statements to our code
- better option
  - considering abstraction and reuse of code
  - is to use a proxy for such logging

JavaScript - Proxy

custom proxy for logging - part 1

- to improve our code reuse and abstraction
  - we may define a proxy for logging within an app.
- e.g.
  - define a custom function, which accepts a `target` object
  - returns a new Proxy object with a getter and setter method

```
// logging with proxy - get and set traps defined
function logger(target) {
  return new Proxy(target, {
    get: (target, property) => {
      console.log(`property read - ${property}`);
      return target[property];
    },
    set: (target, property, value) => {
      console.log(`value '${value}' added to ${property}`);
      target[property] = value;
    }
```

```
  });
}
```

- this is a custom logger
  - wraps passed target object in a proxy with defined getter and setter methods

---

JavaScript - Proxy

custom proxy for logging - part 2

- we may then use this custom function as follows

```
// test object
let planet = {
  name: 'mercury'
};

// new planet object with proxy
planetLog = logger(planet);

// test getting - value for property returned by getter in logger() method...
console.log('default get = ', planetLog.name);

// test setting - value for property set against object
planet.code = 'Me';
```

- in this example
  - we define the initial object
  - then create a new object with a proxy wrapper
- this proxy includes the necessary logger
  - set for both the setter and getter methods
- as we read a property
  - the `get` method will log access and return the requested data
- as we set data
  - we log this update, and then update the target

JavaScript - Proxy

custom proxy for measuring performance - part 1

- another appropriate use of a Proxy is to test performance for a given function
- we may wrap a function with a Proxy, and then `apply` a trap
- this trap may include a simple timer
  - or perhaps a detailed series of tests for the pass function
- e.g.
  - the following function simply loops through a passed counter
  - outputs a series of characters for each iteration

```
// FN: test loop to output to terminal
function loopOutput(counter, marker = '-') {
  if (!counter) {
    return false;
  }
```

```javascript
  // loop through passed counter - check number for even...
  for (i = 0; i <= counter; i++) {
    // check for even counter value
    if (i % 2 === 0) {
      process.stdout.write('+');
    } else {
      // console.log(marker);
      process.stdout.write(marker);
    }
  }
  console.log('\n');
  return true;
}
```

JavaScript - Proxy

custom proxy for measuring performance - part 2

- we may then wrap this function inside a Proxy
  - adding a simple timer for the duration of the loop

```javascript
// wrap function inside custom Proxy
loopTest = new Proxy(loopOutput, {
  // apply simple timer to loop function
  apply: (target, thisArg, args) => {
    console.time("loopTest");
    /* invokes target function - thisArg defines the `this` value
    * if no `thisArg`, undefined will be used instead...
    * thisArg = value to use as `this` when executing a callback
    * args passed to target function loopOutput
    */
    const result = target.apply(thisArg, args);
    console.timeEnd("loopTest");
    return result;
  }
});
```

- `apply` property trap means function value will be executed each time `loopOutput` function is called
- handler will now be executed on function invocation for `loopTest`

JavaScript - Proxy

custom proxy for measuring performance - part 3

- we may then execute this function with its Proxy

```javascript
// call function with counter value and custom marker...
loopTest(75, '-');
```

- markers are output to the terminal
  - includes a record of the loop's performance in milliseconds
- benefit of this approach

- we do not need to modify the original function, `loopOutput`
    - the return, logic, computation &c. will all remain the same
- customisation in this example does not affect the passed function
    - performance checking using the `apply` trap
- `loopOutput` function is now routed through the custom proxy each time it is executed

---

JavaScript - Proxy

custom proxy for property autopopulate

- a proxy may also be used to autopopulate properties
- e.g.
    - we might need to model a directory structure for a file save
    - will require verification of a defined file path
    - or creation of directories to ensure a path may be completed successfully
- latter option may be achieved using a custom proxy
    - create missing directories in a defined path structure
- e.g.

```
// FN: recursive check for dir path and file...
function Directory() {
  return new Proxy({}, {
    get: (target, property) => {
      console.log(`reading property...${property}`);
      // check if property already exists
      if (!(property in target)) {
        // if not - simply add a new directory to target
        target[property] = new Directory();
      }
      // otherwise return property as is from target
      // - write method not implemented for actual directory...
      return target[property];
    }
  });
}

// create new Proxy for function
const rootDir = new Directory();

try {
  // check properties relative to root dir...
  rootDir.testDir.test2Dir.testFile = "test.md";
  console.log('exception not raised...');
} catch (event) {
  // error handling for null exception should be OK due to custom proxy...
  console.log(`exception raised...${event}`);
}
```

JavaScript - Proxy

Reflect a proxy - intro

- ES6 introduced a complement to Proxy usage

- a new built-in object, *Reflect*
- Proxy traps are mapped one-to-one in the Reflect API
- allows an easy combination of Proxy and Reflect usage
- e.g. for each trap there is a matching reflect method

JavaScript - Proxy

Reflect a proxy - `get` trap

- e.g. use `Reflect.get` to define default behaviour for a Proxy getter.

```
const handler = {
    get(target, key) {
        if (key.startsWith('_')) {
           throw new Error(`Property "${ key }" is inaccessible.`)
        }
        return Reflect.get(target, key)
    }
}

const target = {}
const proxy = new Proxy(target, handler)
proxy._secret
```

- in this example, now unable to access the `_secret` property
- obvious benefit of this Reflect usage is the abstraction of `get` usage
  - from Proxy getter to a default, re-usable Reflect `get` method
- use the Proxy getter
  - e.g. to check against data, type &c. in the target
  - then call the Reflect `get` method if successful
- a useful option for restricting access to certain properties through a Proxy
- expose the Proxy instead of the underlying object
  - setting access privileges according to requirements
- if successful, a request will then be handled by the Reflect API method
- access must now go through the Proxy
  - and meet its rules and requirements

JavaScript - Proxy

Reflect a proxy - false return

- returning an error may still be an indication that the `_secret` property exists
- alternative is to return an explicit `false` boolean value for requested hidden property

```
const handler = {
    get(target, key) {
        if (key.startsWith('_')) {
           return false;
        }
        return Reflect.get(target, key)
    }
};
```

```
const library = {
    archive : 'waldzell',
    curator : 'knechts',
    _secret : true
};
const proxy = new Proxy(library, handler);
console.log(`secret = ${proxy._secret}`);
console.log(`archive = ${proxy.archive}`);
```

- a request for underscore value names may still be checked using

```
// _secret is not a private property in object -
console.log(proxy.hasOwnProperty('_secret'))
```

- *underscore* property names are still not private
  - remain visible to specific property checks

JavaScript - Proxy

Reflect a proxy - set trap - part 1

- we may also apply reflection to `set` traps
- reflected `set` method defines behaviour for a setter on a given Proxy object
- equivalent to the default behaviour for the proxy
- e.g.

```
set(target, key, value) {
  return Reflect.set(target, key, value)
}
```

- also add various checks for the passed key...

JavaScript - Proxy

Reflect a proxy - set trap - part 2

- now update our previous example to include a `set` trap with Proxy support

```
const handler = {
    get(target, key) {
        if (key.startsWith('_')) {
           // return false to show prop doesn't exist...
           return false;
        }
        return Reflect.get(target, key)
    },
    set(target, key, value) {
        return Reflect.set(target, key, value);
    }
};
```

- then test property access using the `get` and `set` traps

```
const library = {};
const proxy = new Proxy(library, handler);
proxy.archive = 'mariafels';
proxy._secret = true;
```

JavaScript - Proxy

Reflect a proxy - defaults and checks

- as we use the Reflect object as the default for traps
    - we may add checks, updates &c. to the Proxy trap itself
- e.g. we might add a conditional check to the Proxy
    - then pass a successful update or query to the Reflect method
- default Reflect method allows abstraction for traps from the Proxy
- e.g. we might update each trap with a call to the following conditional check

```
function keyCheck(key, action) {
    if (key.startsWith('_')) {
        throw new Error(`${action} action is not permitted on '${ key }'`)
     }
}
```

- function is called in each trap before continuing to the Reflect method for `get` or `set`

JavaScript - Proxy

proxy wrapper - part 1

- to ensure we restrict access to a `target` object to the defined proxy and reflect traps
    - need to wrap the `target` itself in a Proxy
- target object may have been accessed directly in certain contexts
    - might be beneficial for an admin mode and access
- to restrict access
    - wrap such objects in the Proxy to restrict access to the defined traps and handlers

---

JavaScript - Proxy

proxy wrapper - part 2

- e.g. we can modify our previous example for `get` and `set` traps

```
function proxyWrapper() {
    const target = {};
    const handler = {
        get(target, key) {
            if (key.startsWith('_')) {
               // return false to show prop doesn't exist...
               return false;
            }
            return Reflect.get(target, key)
        },
        set(target, key, value) {
            return Reflect.set(target, key, value);
        }
    };
```

```
    return new Proxy(target, handler);
}
```

JavaScript - Proxy

proxy wrapper - part 3

- `target` may now be accessed and managed using an instantiated proxy

```
const proxiedObject = proxyWrapper();
// set prop & value on target using proxy set trap
proxiedObject.archive = 'waldzell';
// target accessible using proxy get trap
console.log(`target archive = ${proxiedObject.archive}`);
```

- `target` may not be accessed directly using standard property access

```
// target not directly accessible
console.log(`target = ${target}`);
```

JavaScript - Proxy

proxy wrapper - pass object to wrapper

- we may modify this wrapper to also accept an existing object
  - may then be returned wrapped in a Proxy
- e.g.

```
const archive = {
    name: 'waldzell'
}

const proxiedArchive = proxyWrapper(archive);
```

JavaScript - Proxy

proxy wrapper - check object - part 1

- add a further check to ensure we always have a target object to work with..
  - regardless of passed argument value
- e.g. add a check to the `proxyWrapper` function to ensure target is always an object

```
// check object & return empty object if necessary...
function checkTarget(original) {
    // check for existing target object
    if (original.typeof !== 'object' || original === undefined) {
        console.log('not object...');
        const target = {};
        return target;
    } else {
        const target = original;
        return target;
```

```
    }
}
```

JavaScript - Proxy

proxy wrapper - check object - part 2

- if we pass a string instead of a target object
  - we can now create a proxy wrapper with an empty object

```
const proxiedArchive = proxyWrapper('archives');
// set prop & value on target using proxy set trap
proxiedArchive.admin = 'knechts';
proxiedArchive._secret = '1235813';
```

- properties for `admin` and `_secret` may now be set against an empty object
  - due to the passed `archives` string
- we can call this function at the top of the `proxyWrapper` function

```
function proxyWrapper(original) {
    // check target for proxy wrapper - original must be object
  const target = checkTarget(original);
  ...
}
```

JavaScript - Proxy

proxy wrapper - update property access check

- also abstract initial check for property access using a defined character delimiter
- e.g.

```
// check property access using defined char delimiter
function checkDelimiter(key, char) {
    // check key relative to specified char delimiter
    if (key.startsWith(char)) {
        // return false to show prop not available
        return true;
     }
}
```

- simply check defined delimiter character relative to passed property key
  - may then be called in the `proxyWrapper` function

```
if (checkDelimiter(key, '_')){
  return false;
}
```

JavaScript - Proxy

proxy wrapper - restricting access

- in the previous examples

    - we define the `target` object both inside and outside the `proxyWrapper` function
- both may be effective options for restricting object access depending upon context
- internal object declaration for `target` restricts full access to the Proxy object
- any traps for the object will only be accessible using the Proxy object
- consumer must use the instantiated Proxy object to read, write, query &c.
- external `target` object may still be useful after it has been wrapped by a Proxy object
- restricted access is controlled by only exposing the target as a Proxy object
- e.g. if we exposed the target as an access point for a pubic API
    - proxy object will be exposed and not the original target object

---

JavaScript - Proxy

proxy and schema validation

- objects may be defined for a specific purpose or context
    - requires control over stored properties and values
- validation allows us define the structure of an object
    - e.g. its properties, types, permitted values &c.
- we may use a third party module or custom function
    - may return an error for invalid input and data…
- still need to ensure that the object storing the input data is restricted
    - e.g. to authorised access both internal and external to the app
- another option is to use a Proxy with validation of the object
    - proxy object may be used to provide access to the model object for validation
- another benefit of a proxy with validation is the separation of concerns
    - data object remains separate from the validation
- consumer never accesses the input object directly
    - given a proxy object with validation checks and balances
- original input object remains a plain object due to nature of Proxy object usage
- defined proxy handlers for validation &c. may also be referenced and reused
    - reuse across multiple Proxies…

JavaScript - Proxy

proxy and validator - part 1

- create an initial validator
    - using a Proxy, a map, and defined handlers for required object properties
- e.g. as a property is set through a proxy object
    - its key may be checked against the map
    - if there is a rule for the key, its *handler* value will be executed
    - handler executed to check that the property is valid

```
// MAP - validation rules for properties
const validationMap = new Map();

// TRAPS - define traps for proxy
const validator = {
    // set trap
    set(target, key, value) {
        // check map for matching handler
        if (validationMap.has(key)) {
            // return handler function if available...pass value as parameter
```

```
            return validationMap.get(key)(value);
        }

        // else - default reflect set method for proxy
        return Reflect.set(target, key, value);
    }
};
```

JavaScript - Proxy

proxy and validator - part 2

- value may be passed as a parameter to the handler function
  - stored in the map for the requested key
  - function may include a validation, check &c.

```
// RULES - define executable rules for permitted object properties
// e.g. log, update state, get state, broadcast, subscribe...
// e.g. sample validation for text to log
function validateLog(text) {
    if (typeof text === 'string') {
        console.log(`logger = ${text}`);
    } else {
        throw new TypeError(`logger requires text input...`);
    }
}
```

JavaScript - Proxy

proxy and validator - part 3

- we may then use this proxy and map as follows

```
// set key and handler function in map
validationMap.set('logger', validateLog);
// empty object to wrap with proxy
const process = {};
// instantiate proxy object
const proxyProcess = new Proxy(process, validator);

// string set using handler for logger
proxyProcess.logger = 'test string = hello proxy...';
// number will not be set - fails validation
proxyProcess.logger = 96;
```

Project Outline - Setup & Usage

intro

- consider task runners and build tools
  - e.g. Grunt, Webpack...
  - relative to build distributions and development environments

- for a new project, begin by initialising a *Git* repository
  - initialise in the root directory
- also add a `.gitignore` file to our local repository
  - define files and directories not monitored by Git's version control
- then initialise a new NodeJS based project using *NPM*
  - execute the following terminal command

```
npm init
```

- answer initial `npm init` questions or use suggested defaults
- `package.json` file created
  - default metadata may be updated as project develops

Project Outline - Setup & Usage

directory structure - part 1

- basic project layout may follow a sample directory structure,

```
.
|-- build
|   |-- css
|   |-- img
|   |-- js
|-- src
|   |-- assets
|   |-- css
|   |-- js
|   |__ app.js
|-- temp
|-- testing
|__ index.html //applicable for client-side, webview apps &c.
```

- sample needs to be modified relative to a given project
- `build` , `temp` , and `testing` will include files and generated content
  - from various build tasks
- `build` and `temp` directories may be created and cleaned automatically
  - as part of the build tasks
  - do not need to be created as part of the initial directory structure

---

Project Outline - Setup & Usage

directory structure - part 2

- example structure adds `index.html` file to root of project structure
  - e.g. for client-side and webview based development
- structure includes `build` directories
  - may not add until build tasks for a `release` distribution
  - commonly include bundling, minification, uglifying, &c.
- `build` directory will be part of a build task
- also update our project's `.gitignore` file

```
.DS_Store
node_modules/
```

```
*.log
build/
temp/
```

Project Outline - Setup & Usage

install and configure Grunt

- start by installing and configuring Grunt for the above sample project structure

```
npm install grunt --save-dev
```

- install assumes a global scope for the NPM package `grunt-cli`
  - saves metadata to `package.json` for development builds only
- to use Grunt with a project
  - add a config file, `Gruntfile.js` to the project's root directory
  - includes initial exports for tasks and targets
- we may then load and register the required tasks

Project Outline - Setup & Usage

`Gruntfile.js` - initial exports

- Grunt config is again dependent on specifics of the project
- we may add some common options
  - e.g. linting, build distributions, minification and bundling, uglifying, sprites &c.
- use of `rollup` will depend upon required support for modules
  - including ES modules within JavaScript apps

```
module.exports = function(grunt) {
    grunt.initConfig(
        {
            jshint: {
                all: ['src/**/*.js'],
                options: {
                    'esversion': 6,
                    'globalstrict': true,
                    'devel': true,
                    'browser': true
            }
            },
            rollup: {
                release: {
                options: {},
                files: {
                'temp/js/rolled.js': ['src/js/main.js'],
                },
                }
        },
            uglify: {
                release: {
                    files: {
                        'build/js/mini.js': 'temp/js/*.js'
```

```
                    },
                }
            },
            sprite: {
                release: {
                    src: 'src/assets/images/*',
                    dest: 'build/img/icons.png',
                    destCss: 'build/css/icons.css'
                }
            },
            clean: {
            folder: ['temp'],
            }
        }
    );

};
```

Project Outline - Setup & Usage

`Gruntfile.js` **- custom task**

- we may add custom tasks such as metadata generation,

```
buildMeta: {
    options: {
        file: './meta.md',
        developer: 'debug tester',
        build: 'debug'
    }
},
```

- we may add tasks for CSS &c. as we continue to develop the project

---

Project Outline - Setup & Usage

`Gruntfile.js` **- use tasks - part 1**

- after defining the exports for tasks and targets,
  - we can load the required Grunt plugin modules
  - register the required tasks
  - ...
- we may run these registered tasks together
  - or separately relative to distribution and environment
- e.g. load the plugins for the required tasks,

```
// linting, module bundling, minification, directory cleanup...
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-rollup');
grunt.loadNpmTasks('grunt-contrib-uglify-es');
grunt.loadNpmTasks('grunt-spritesmith');
grunt.loadNpmTasks('grunt-contrib-clean');
```

Project Outline - Setup & Usage

`Gruntfile.js` - use tasks - part 2

- plugins correspond to installed NPM packages for current project
  - e.g.

```
npm install grunt-contrib-jshint --save-dev
npm install grunt-rollup --save-dev
npm install grunt-contrib-uglify-es --save-dev
npm install grunt-spritesmith --save-dev
npm install grunt-contrib-clean --save-dev
```

Project Outline - Setup & Usage

`Gruntfile.js` - register custom task

- we may then register a custom task for various targets in the builds
  - e.g.

```
// custom task - build meta for default debug
grunt.registerTask('buildMeta', function() {
    console.log('debug build...');
    const options = this.options();
    metaBuilder(options);
});

//custom task - build meta for release
grunt.registerTask('buildMeta:release', function() {
    console.log('release build...');
    // define task options - incl. defaults
    const options = this.options({
        file: 'build/release_meta.md',
        developer: "spire & signpost",
        build: "release"
    });
    metaBuilder(options);
});
```

Project Outline - Setup & Usage

`Gruntfile.js` - register builds

- then register some build tasks
  - tasks may combine the options from the config
  - provides the execution of staggered tasks for a single build call
- e.g. a debug build may include
  - linting, custom metadata, and a clean task

```
// debug build tasks - default tasks during development...
grunt.registerTask('build:debug', ['jshint', 'buildMeta', 'clean']);
```

- we may also define a build process for staging or release

```
// build tasks with specific 'release' targets...
grunt.registerTask('build:release', ['jshint', 'rollup:release', 'uglify:release', 'sprite:release', 'bu
```

- we may run and test Grunt for the current project
  - relative to project requirements, e.g. debug or release

```
grunt build:debug
```

- or

```
grunt build:release
```

Project Outline - Setup & Usage

development with environments

- as we develop more complex apps
  - need to consider how we configure and use such build tools
- e.g. with various environments
  - development
  - staging
  - production / release
- we can define a *debug* or *release* distribution build
  - use with each of these environments

Project Outline - Setup & Usage

environment setup - development - part 1

- app development will primarily focus on a `debug` distribution
  - provide tasks such as linting, testing, metadata, watch, &c.
  - becomes common distribution for active, ongoing development
- also need to ensure environment variables are aggregated
  - allows the app to run as expected
  - stored in the same manner regardless of `debug` or `release`
- difference is use of encryption
  - and the nature of the required environment configs
- bundling with minification and uglifying
  - usually added to a project as part of `release` distribution
  - may serve little practical benefit for ongoing active development

---

Project Outline - Setup & Usage

environment setup - development - part 2

- we may define a common structure for Node based apps as follows

```
.
|-- debug
|-- src
|   |-- assets
|   |-- js
```

```
|-- temp
|-- testing
|__ app.js
```

- develop the app, including the app source code, in the `src` directory
- build our app in the `debug` directory
  - each time we need to check and debug usage
- temporary build artifacts may be added to the `temp` directory
  - cleaned after each build workflow has been completed
- e.g. each time we complete a call to `build:debug`
  - clean, where applicable, the build artifacts
- we may also choose to combine `debug` and `temp`
  - a single `temp` directory
  - depending upon project requirements

Project Outline - Setup & Usage

environment setup - development - part 3

- for a client-side or mobile hybrid app
  - slightly modify this directory structure, e.g.

```
.
|-- debug
|    |-- css
|    |-- img
|    |-- js
|-- src
|    |-- assets
|    |-- css
|    |-- js
|    |__ app.js
|-- temp
|-- testing
|__ index.html
```

- `assets` directory may include raw image files, icons, &c.
- test builidng these image assets as sprites
  - added to the `img` directory during the build
- also use *image optimisation* at this stage
  - e.g. test UI and UX performance
- part of the `debug` distribution is the use of `watch` for live reloading
  - `nodemon` for Node.js based apps
- also consider tasks to aggregate logging within the app's code
- may include explicit `console.log()` statements, and error handling

Project Outline - Setup & Usage

environment setup - development Grunt config - part 1

- update our Grunt config
  - use a `debug` distribution in current *development* environment

- e.g. add any required build options for debug
  - then integrate required environment config variables &c.
- start with *unencrypted JSON* files
- may contain defaults for options
  - e.g. current environment, server's port number &c.

```
{
    "NODE_ENV": "development",
    "PORT": 3826
}
```

Project Outline - Setup & Usage

environment setup - development Grunt config - part 2

- define some additional project directories
  - e.g. encrypted and decrypted config files

```
.
|-- env
|   |-- defaults
|   |-- private
|   |-- secure
```

- `env/defaults` contains the unencrypted defaults
  - as defined in `defaults.json`
- `env/private` includes decrypted secure files
- `env/secure` should be reserved for encrypted files
  - we may add to version control
- `env/private` should **not** be commited to version control
- a few different options for file encryption
  - e.g. RSA based public/private keys, GNU Privacy Guard (GPG, or GnuPG)
- further details in the extra notes
  - encryption, signatures, and verification of files
  - includes step by step examples for working with RSA
  - and extra layers of verification for a file with generated signatures

---

Project Outline - Setup & Usage

merging config sources

- as a project develops, we may produce various sources of configuration
- may include sources such as
  - JSON files
  - JavaScript objects
  - environment variables
  - process arguments
  - ...
- to help merge such disparate config sources
  - add an NPM module such as `nconf`
  - nconf
- or we may simply load environment variables
  - e.g. from a project's `.env` file using the package `dotenv`

  - dotenv

Project Outline - Setup & Usage

sample waterfall with nconf

- with `nconf` we may bundle various config stages for a project
  - e.g.

```
const nconf = require('nconf');
nconf.argv();
nconf.env();
nconf.file('dev', 'development.json');
module.exports = nconf.get.bind(nconf);
```

- getting config variables and settings from defined stores in defined cascading order
- order is prioritised
  - allowing overrides and defaults at various stages of the cascade
  - e.g. if a value is given in the command arguments, `argv`

Project Outline - Setup & Usage

continuous development

- continuous development (CD)
  - allows a developer to work on app code &c. without many customary interruptions
  - e.g. server reboots, code refreshes, debugging, linting &c.
- CD often reduces repetitive tasks in a development flow
  - helping to automate processes and development
- build process may be automated and run whenever a pertinent change is detected

---

Project Outline - Setup & Usage

continuous development - add a `watch` task - part 1

- add a *watch* task to a build flow
  - allow a rebuild each time a given file is edited and then saved
- e.g. for Grunt, we may add the plugin module `grunt-contrib-watch`

```
npm install grunt-contrib-watch --save-dev
```

- and update the Grunt config

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

- plugin watches file system for code changes in a tracked project
  - then runs the affected tasks as required
- basic `watch` example might include the following

```
watch: {
    js: {
        tasks: ['jshint:client'],
        files: ['src/**/*.js']
    }
}
```

- continuously checks `src` directory for JavaScript file change or addition
  - then runs the `jshint:client` task
- this type of `watch` provides a broad approach to managing project changes

---

Project Outline - Setup & Usage

continuous development - add a `watch` task - part 2

- then include additional *targets* relative to project requirements
  - e.g. add further JS specific targets, CSS, sprites &c.
- we may also define separate build tasks to use `watch`
  - e.g.

```
// dev tasks - combine debug with watch
grunt.registerTask('dev', ['build:debug', 'watch']);
```

- which we may call as follows,

```
grunt dev
```

- executes the tasks for `build:debug`
- then starts *watching* the specified targets

Project Outline - Setup & Usage

continuous development - live reload - part 1

- also use `watch` to add support for *live reloads*
- built-in support with the `grunt-contrib-watch` plugin
- reload option uses *web sockets*
  - originally designed for browser based real-time communication and synchronisation
- LiveReload option listens for changes to monitored files, directories &c.
  - then reload and refresh the current active app
- support for the LiveReload task may added as follows

```
livereload: {
    options: {
        livereload: true
    },
    files: ['build/**/*', './*.html'],
},
```

- provides a live reload server - usually runs at `localhost:35729`
- object includes a property to confirm `livereload`
  - then defines files to watch to initiate a reload
- e.g. in this example
  - watching `build` directory, its children, then the root directory for any HTML files
  - includes any changes to default `index.html` file
- *n.b.* this server does not actually reload the app for us
  - need to use a server to host the app
  - host server is monitoring this `livereload` server

Project Outline - Setup & Usage

continuous development - live reload - part 2

- `livereload` also provides a setup script for the test app
- two common options for use
  - add a link to this script in our project's `index.html` file

```
<script src="http://localhost:35729/livereload.js"></script>
```

- or
  - use a Grunt plugin, `grunt-contrib-connect`
- grunt-contrib-connect
  - automatically injects script in our app's code
  - preferred option for ongoing development
- install this plugin as follows

```
npm install grunt-contrib-connect --save-dev
```

- then update the `Gruntfile.js` config

```
connect: {
    server: {
        options: {
          port: 8080,
          base: '.',
          hostname: '*',
          protocol: 'http',
          livereload: true,
        }
    },
},
```

Project Outline - Setup & Usage

continuous development - live reload - part 3

- need to update the required build tasks to use these plugins
  - e.g. add connect and livereload support to `dev` build task

```
// dev tasks - combine debug with watch, live server, and live reload
grunt.registerTask('dev', ['build:debug', 'connect', 'watch']);
```

- then run this build task

```
grunt dev -v
```

- `-v` flag outputs verbose messages
  - helps initially check everything is running as expected

Project Outline - Setup & Usage

add CSS support - part 1

- app styles will, customarily, include a combination of options
  - e.g. CSS stylesheets and dynamic JavaScript based style properties
- to work with CSS stylesheets, similar to JavaScript files

- consider a Grunt task for minifying these files
- we need to install the Grunt module, `grunt-contrib-cssmin`

```
npm install grunt-contrib-cssmin --save-dev
```

- then add the following to include this package in the `Gruntfile.js` config

```
grunt.loadNpmTasks('grunt-contrib-cssmin');
```

- and update the build task for a release distribution

```
// build tasks with specific 'release' targets...
  grunt.registerTask('build:release', ['rollup:release', 'cssmin:release', 'uglify:release', 'buildMeta
```

- referencing the following task for `cssmin`

```
cssmin: {
    release: {
        options: {
        banner: '/* minified css file - basic-es-modules */'
        },
        files: {
      'build/css/mini.css': [
        'src/css/main.css',
        ]
        }
    }
},
```

Project Outline - Setup & Usage

add CSS support - part 2

- with the minified CSS stylesheet built
  - add a link to this stylesheet in the `index.html` file

```
<!-- css styles - main -->
<link rel="stylesheet" href="./build/css/mini.css">
```

- then update the `watch` task by adding the following for CSS

```
css: {
    files: ['src/**/*.css'],
    tasks: ['cssmin:release']
},
```

- then run the usual Grunt build tasks
  - e.g. to minify the CSS stylesheets, and watch for any updates and changes…

Project Outline - Setup & Usage

Watch update

- current `watch` task includes support for CSS, JS, and HTML
- includes checks for modifications
  - e.g. to any defined `src` directories for CSS and JS
  - monitors any HTML files in the app's root directory

- a working `watch` task is as follows

```
watch: {
    js: {
        files: ['src/**/*.js'],
        tasks: ['jshint:client', 'rollup:release', 'uglify:release']
    },
    css: {
        files: ['src/**/*.css'],
        tasks: ['cssmin:release']
    },
    html: {
        files: ['./*.html']
    },
    livereload: {
        options: {
            livereload: true
        },
        files: ['build/**/*', './*.html'],
    },
},
```

JS Server-side considerations - save data

save JSON in travel notes app

- need to be able to save our simple notes
- now load from a JSON file as the app starts
  - also we can add new notes, delete existing notes…
- not as simple as writing to our existing JSON file direct from JS
  - security implications if that was permitted directly from the browser
- need to consider a few server-side options
- could use a combination of PHP on the server-side
  - with AJAX jQuery on the client-side
  - traditional option with a simple ajax post to a PHP file on the server-side
- consider JavaScript options on the client and server-side
- brief overview of working with **Node.js**

Server-side considerations - intro

- normally define computer programs as either client-side or server-side programs
- server-side programs normally abstract a resource over a network
  - enabling many client-side programs to access at the same time
  - a common example is file requests and transfers
- we can think of the client as the web browser
- a web server as the remote machine abstracting resources
- abstracts them via **hypertext transfer protocol**
  - HTTP for short
- designed to help with the transfer of HTML documents
  - HTTP now used as an abstracted wrapper for many different types of resources
  - may include documents, media, databases…

Image - Client-side and server-side computing

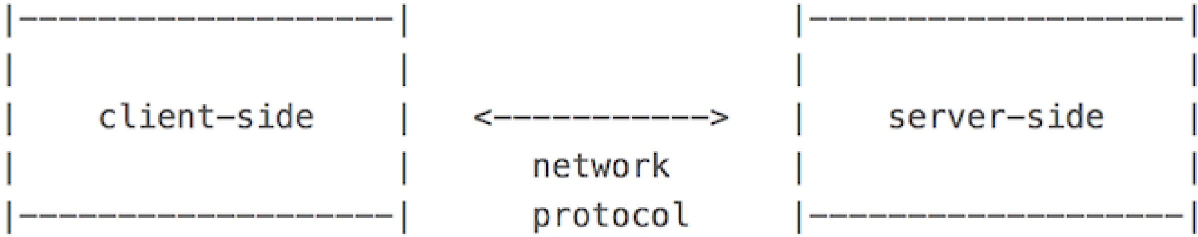


Figure 8: client-side & server-side

Server-side considerations - Node.js

intro - what is Node.js?

- Node.js is, in essence, a JavaScript runtime environment
    - designed to be run outside of the browser
- designed as a general purpose utility
- can be used for many different tasks including
    - asset compilation
    - monitoring
    - scripting
    - web servers
- with Node.js, role of JS is changing
    - moving from client-side to a support role in back-end development

---

Server-side considerations - Node.js

intro - speed of Node.js

- a key advantage touted for Node.js is its speed
- many companies have noted the performance benefits of implementing Node.js
    - including PayPal, Walmart, LinkedIn…
- a primary reason for this speed boost is the underlying architecture of Node.js
- Node.js uses an **event-based** architecture
- instead of a threading model popular in compiled languages
- Node.js uses a single event thread by default
- all I/O is asynchronous

Server-side considerations - Node.js

intro - conceptual model for processing in Node.js

- how does Node.js, and its underlying processing model, actually work?

- client sends a hypertext transfer protocol, HTTP, request
  - request or requests sent to Node.js server
- event loop is then informed by the host OS
  - passes applicable request and response objects as JavaScript closures
  - passed to associated worker functions with callbacks
- long running jobs continue to run on various assigned worker threads
- responses are sent from the non-blocking workers back to the main event loop
  - returned via a callback
- event loop returns any results back to the client
  - effectively when they're ready

Image - Client-side and server-side computing

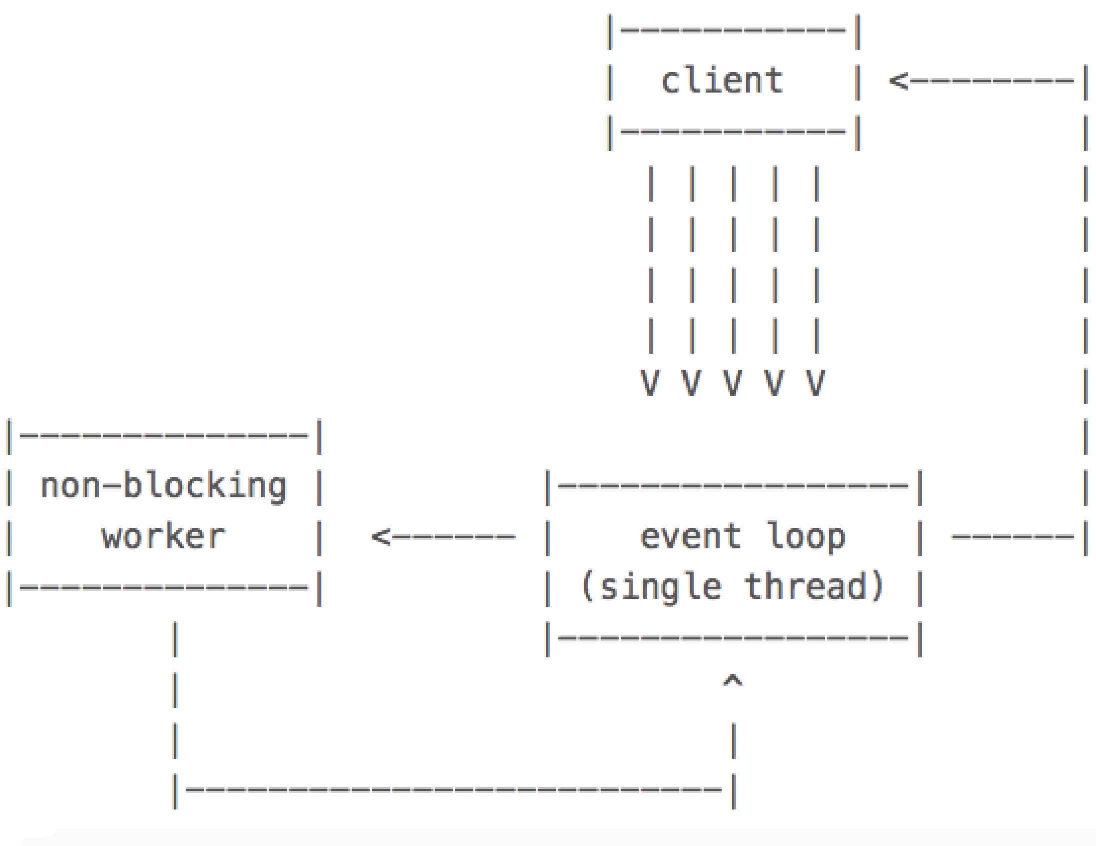


Figure 9: Node.js - conceptual model for processing

Server-side considerations - Node.js

intro - threaded architecture

- concurrency allows multiple things to happen at the same time
- common practice on servers due to the nature of multiple user queries
- Java, for example, will create a new thread on each connection

- threading is inherently resource expensive
- size of a thread is normally around 4MB of memory
- naturally limits the number of threads that can run at the same time
- also inherently more complicated to develop platforms that are thread-safe
  - thereby allowing for such functionality
- due to this complexity
  - many languages, eg: Ruby, Python, and PHP, do not have threads that allow for real concurrency
  - without custom binaries
- JavaScript is similarly single-threaded
  - able to run multiple code paths in parallel due to **events**

Server-side considerations - Node.js

intro - event-driven architecture

- JavaScript originally designed to work within the confines of the web browser
- had to handle restrictive nature of a single thread and single process for the whole page
- synchronous blocking in code would lock up a web page from all actions
  - JavaScript was built with this in mind
- due to this style of I/O handling
  - Node.js is able to handle millions of concurrent requests on a single process
- added, using libraries, to many other existing languages
  - Akka for Java
  - EventMachine for Ruby
  - Twisted for Python
  - …
- JavaScript syntax already assumes events through its use of callbacks
- **NB:** if a query etc is CPU intensive instead of I/O intensive
  - thread will be tied up
  - everything will be blocked as it waits for it to finish

Server-side considerations - Node.js

intro - callbacks

- in most languages
  - send an I/O query & wait until result is returned
  - wait before you can continue your code procedure
- for example, submit a query to a database for a user ID
  - server will pause that thread/process until database returns result for ID query
- in JS, this concept is rarely implemented as standard
- in JS, more common to pass the I/O call a **callback**
- in JS, this **callback** will need to run when task is completed
  - eg: find a user ID and then do something, such as output to a HTML element
- biggest difference in these approaches
  - whilst the database is fetching the user ID query
  - thread is free to do whatever else might be useful
  - eg: accept another web request, listen to a different event…
- this is one of the reasons that Node.js returns good benchmarks and is easily scaled
- **NB:** makes Node.js well suited for I/O heavy and intensive scenarios

---

Server-side considerations - Node.js

install Node.js

- a number of different ways to install **Node.js**, **npm**, and the lightweight, customisable web framework **Express**
- run and test Node.js on a local Mac OS X or Windows machine
- download and install a package from the following URL
  - Node.js - download
- install the Node module, **Express**
- Express is a framework for web applications built upon Node.js
  - minimal, flexible, & easily customised server
- use *npm* to install the Express module

```
npm install -g express
```

- `-g` option sets a global flag for Express instead of limited local install
- installs Express command line tool
  - allows us to start building our basic web application
- now also necessary to install Express application generator

```
npm install -g express-generator
```

Server-side considerations - Node.js

NPM - intro

- **npm** is a package manager for Node.js
- Developers can use **npm** to share and reuse modules in Node.js applications
- **npm** can also be used to share complete Node.js applications
- example modules might include
  - Markup, YAML etc parsers
  - database connectors
  - Express server
  - ...
- **npm** is included with the default installers available at the Node.js website
- test whether **npm** is installed, simply issue the following command

```
npm
```

- should output some helpful information if **npm** is currently installed
- **NB:** on a Unix system, such as OS X or Linux
  - best to avoid installing **npm** modules with `sudo` privileges

---

Server-side considerations - Node.js

NPM - installing modules

- install existing **npm** modules, use the following type of command

```
npm install express
```

- this command installs module named `express` in the current directory
- it will act as a local installation within the current directory
- installing in a folder called `node_modules`
  - this is the default behaviour for current installs

- we can also specify a global install for modules
  - eg: we may wish to install the **express** module with global scope

```
npm install -g express
```

- again, the `-g` flag specifies the required global install

Server-side considerations - Node.js

NPM - importing modules

- import, or effectively add, modules in our Node.js code
  - use the following declaration

```
var module = require('express');
```

- when we run this application
  - Node.js looks for the required module library and its source code

---

Server-side considerations - Node.js

NPM - finding modules

- official online search tool for **npm** can be found at
  - npmjs
- example packages include options such as
  - browserify
  - express
  - grunt
  - bower
  - karma
  - ...
- also search for Node modules directly
  - search from the command line using the following command

```
npm search express
```

- returns results for module names and descriptions

Server-side considerations - Node.js

CommonJS modules - custom design and usage

- extra notes available on CommonJS module usage
  - custom design and usage
  - library structure and development
- extra source code examples available
  - general usage
  - custom modules
  - custom library example

Server-side considerations - Node.js

NPM - specifying dependencies

- ease Node.js app installation
  - specify any required dependencies in an associated `package.json` file
- allows us as developers to specify modules to install for our application
  - which can then be run using the following command

```
npm install
```

- helps reduce the need to install each module individually
- helps other users install an application as quickly as possible
- our application's dependencies are stored in one place
- example `package.json`

```
{
"name": "app",
"version": "0.0.1",
"dependencies": {
  "express": "4.2.x",
  "underscore": "-1.2.1"
}
}
```

Server-side considerations - Node.js

initial Express usage

- now use Express to start building our initial basic web application
- Express creates a basic shell for our web application
  - `cd` to working directory and use the following command

```
express /node/test-project
```

- command makes a new directory
  - populates with required basic web application directories and files
- `cd` to this directory and install any required dependencies,

```
npm install
```

- then run our new app,

```
npm start
```

- or run and monitor our app,

```
nodemon start
```

Server-side considerations - Node.js

initial Express server - setup

- we've now tested **npm**, and installed our first module with **Express**
- test **Express**, and build our first, simple server
- initial directory structure

```
|- .
   |- 424-node
      |- node_modules
```

- need to do is create a JS file to store our server code, so we'll add `server.js`

```
|- .
   |- 424-node
      |- node_modules
      |- server.js
```

- start adding our Node.js code to create a simple server

Server-side considerations - Node.js

initial Express server - server.js - part 1

- add some initial code to get our server up and running

```
/* a simple Express server for Node.js*/
var express = require("express"),
    http = require("http"),
    appTest;

// create our server - listen on port 3030
appTest = express();
http.createServer(appTest).listen(3030);

// set up routes
appTest.get("/test", function(req, res) {
  res.send("welcome to the 424 test app.");
});
```

- then start and test this server as follows at the command line

```
node server.js
```

Server-side considerations - Node.js

initial Express server - server.js - part 2

- open our web browser, and use the following URL

```
http://localhost:3030
```

- this is the route of our new server
  - to get our newly created route use the following URL

```
http://localhost:3030/test
```

- this will now return our specified route, and output message
- update our `server.js` file to support root directory level routes

```
appTest.get("/", function(req, res) {
  res.send("Welcome to the 424 server.")
});
```

- now load our server at the root URL

```
http://localhost:3030
```

- stop server from command line using `CTRL` and `c`

Server-side considerations - Node.js

initial Express server - server.js - part 3

- currently, initial Express server is managing some static routes for loading content
  - we simply tell the server how to react when a given route is requested
- what if we now want to serve some HTML pages?
  - Express allows us to set up routes for static files

```
//set up static file directory - default route for server
appTest.use(express.static(__dirname + "/app"));
```

- now defining Express as a static file server
  - enabling us to publish our HTML, CSS, and JS files
  - published from our default directory, `/app`
- if requested file not available
  - server will check other available routes
  - or report error to browser if nothing found
- DEMO - 424-node

Image - Client-side and server-side computing

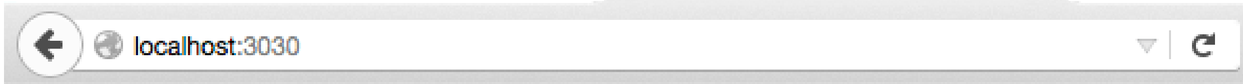
Server-side considerations - Node.js

working with data - JSON

- let us now work our way through a basic Node.js app
- serve our JSON, then read and load from a standard web app
- create our initial `server.js` file

```
var express = require('express'),
    http = require("http"),
    jsonApp = express(),
    notes = {
      "travelNotes": [{
      "created": "2015-10-12T00:00:00Z",
      "note": "Curral das Freiras..."
      }]
    };

jsonApp.use(express.static(__dirname + "/app"));
```



welcome to the app file directory

Figure 10: simple Express server output

```
http.createServer(jsonApp).listen(3030);

//json route
jsonApp.get("notes.json", function(req, res) {
  res.json(notes);
});
```

Image - Client-side and server-side computing

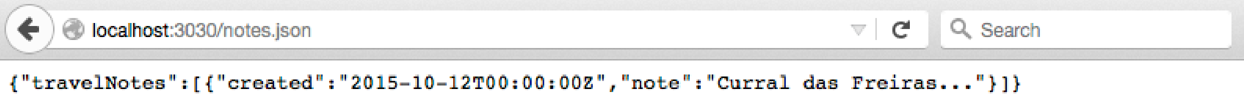


Figure 11: simple Express JSON route output

Server-side considerations - Node.js

working with data - JSON

- now have our `get` routes setup for JSON
- now add some client-side logic to read that route
- render to the browser
- same basic patterns we've seen before
  - using jQuery's `.getJSON()` function

```
...
  $.getJSON("notes.json", function (response) {
    console.log("response = "+response.toSource());
    buildNote(response);
  })
...
```

- response object from our JSON
  - this time from the server and not a file or API
- use our familiar functions to create and render each note
  - call our normal `buildNote()` function
- DEMO - 424-node-json1

Image - Client-side and server-side computing

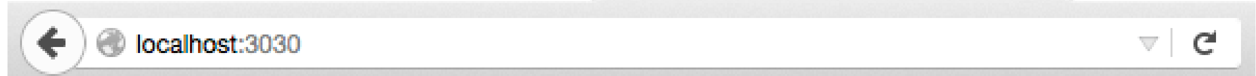


Figure 12: simple Express JSON route output to DOM

Server-side considerations - Node.js

working with data - post data

- we've seen examples that load JSON data
  - using jQuery's `.getJSON()` function
- now consider jQuery's `post` function
  - allow us to easily send JSON data to the server
  - simply called `post`
- begin our updates by creating a new route in our Express server
  - one that will handle the `post` route

```
jsonApp.post("/notes", function(req, res) {
  //return simple JSON object
  res.json({
```

```
    "message": "post complete to server"
  });
});
```

Server-side considerations - Node.js

working with data - post data

- may look similar to our earlier `get` routes
  - difference due to browser restrictions
  - can't simply request direct route using our browser
  - as we did with `get` routes
- need to change JS we use for the client-side
  - allows us to post new route
  - then enables view of the returned message
- update our test app to store data on the server
  - then initialise our client with this stored data

Server-side considerations - Node.js

working with data - post data

- start with a simple check that the post route is working correctly
  - add a button, submit a request to the post route, and then wait for the response
  - add event handler for a button

```
$("#post").on("click", function() {
  $.post("notes", {}, function (response) {
    console.log("server post response returned..." + response.toSource());
  })
});
```

- submit a `post` request
  - specify the route for the post to the Node.js server
  - then specify the data to post - an empty object in this example
  - the specify a callback for the server's response
- test returns the following output to the browser's console,

```
server post response returned...({message:"post complete to server"})
```

Server-side considerations - Node.js

working with data - post data

- now send some data to the server
  - add new note to our object
- update the server to handle this incoming object
  - process the submitted jQuery JSON into a JavaScript object
  - ready for use with the server
- use the **Express** module's `body-parser` plugin
- update `server.js` as follows

```
//add body-parser for JSON parsing etc...
var bodyParser = require("body-parser");
...
//Express will parse incoming JSON objects
jsonApp.use(bodyParser.urlencoded({ extended: false }));
...
```

- as server receives new JSON object
  - it will now parse, or process, this object
  - ensures it can be stored on the server for future use

Server-side considerations - Node.js

working with data - post data

- now update our test button's event handler
  - send a new note as a JSON object
- note will retrieve its new content from the input field
  - gets the current time from the node server

```
$(".note-input button").on("click", function() {
  //get values for new note
  var note_text = $(".note-input input").val();
  var created = new Date();
  //create new note
  var newNote = {"created":created, "note":note_text};
  //post new note to server
  $.post("notes", newNote, function (response) {
    console.log("server post response returned..." + response.toSource());
  })
});
```

- input field and button follow the same pattern as previous examples

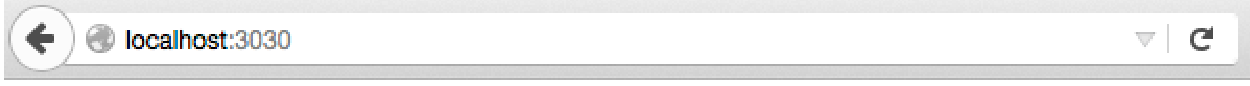
```
<!-- note input -->
<section class="note-input col-6">
    <h5>add note</h5>
    <input><button>add</button>
</section>
```

- DEMO - 424-node-json2

Image - Client-side and server-side computing

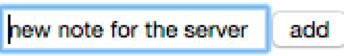
Node.js extras - API examples

- various custom API examples
  - ToDos & ToDos with testing
  - authentication examples
  - Notetaking
  - ...with Socket.io
  - ...



Node and JSON

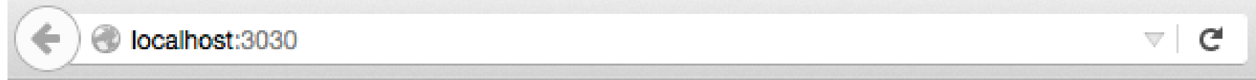
add note



Curral das Freiras...

app's copyright information, additional links...

Figure 13: Node.js and Express - post new note to server



Node and JSON

add note

Curral das Freiras...

new note for the server

app's copyright information, additional links...

Figure 14: Node.js and Express - get new notes from server

- Twitter with Node.js custom server
  - user queries &c.
  - OAuth based login and authentication
- Yelp with Node.js custom server
  - custom server and remote API query
  - sample handling of local API for queries

Server-side considerations - data storage

intro

- tested Node.js, created a server for hosting our files and routes with ExpressJS
  - read JSON from the server
  - updated our JSON on the server-side
- works well as long as we do not need to restart, repair, update etc our server
- data lost with restart etc…
- need to consider a persistent data storage
  - independent from the application
- NoSQL options such as Redis and MongoDB
- integration with Node.js

Server-side considerations - data storage

SQL or NoSQL

- common database usage and storage
  - often thought solely in terms of SQL, or structured query language
- SQL used to query data in a relational format
- relational databases, for example MySQL or PostgreSQL, store their data in tables
  - provides a semblance of structure through rows and cells
  - easily cross-reference, or relate, rows across tables
- a relational structure to map authors to books, players to teams…
  - thereby dramatically reducing redundancy, required storage space…
- improvement in storage capacities, access…
  - led to shift in thinking, and database design in general
- started to see introduction of non-relational databases
  - often referred to simply as **NoSQL**
- with NoSQL DBs
  - redundant data may be stored
  - such designs often provide increased ease of use for developers
- some NoSQL examples for specific use cases
  - eg: fast reading of data more efficient than writing
  - specialised DB designs

Server-side considerations - data storage

Redis - intro

- Redis provides an excellent example of NoSQL based data storage
- designed for fast access to frequently requested data
- improvement in performance often due to a reduction in perceived reliability
  - due to in-memory storage instead of writing to a disk

- able to flush data to disk
  - performs this task at given points during uptime
  - for majority of cases considered an in-memory data store
- stores this data in a **key-value** format
  - similar in nature to standard object properties in JavaScript
- Redis often a natural extension of conventional data structures
- Redis is a good option for quick access to data
  - optionally caching temporary data for frequent access

Server-side considerations - data storage

Redis - installation

- On OS X, use the Homebrew package manager to install Redis

```
brew install redis
```

- Windows port maintained by the Microsoft Open Tech Group - Redis
  - or use Windows package manager - https://chocolatey.org/
  - try WSL
  - **n.b.** Redis on Windows is not recommended...
- for Linux - download, extract, and compile Redis

```
$ wget http://download.redis.io/releases/redis-3.0.5.tar.gz
$ tar xzf redis-3.0.5.tar.gz
$ cd redis-3.0.5
$ make
```

Server-side considerations - data storage

Redis - server and CLI

- start the Redis server with the following command,

```
redis-server
```

- interact with our new server directly using the CLI tool,

```
redis-cli
```

- store some data in Redis using the `set` command
  - create a new key for `notes` , and then set its value to `0`
  - if value is set, Redis returns `OK`

```
set notes 0
```

- retrieve a value using the `get` command
  - returns our set value of `0`

```
get notes
```

Image - Client-side and server-side computing

```
Drs-MacBook-Air-2:~ ancientlives$ redis-cli
127.0.0.1:6379> set notes 0
OK
127.0.0.1:6379> get notes
"0"
127.0.0.1:6379>
```

Figure 15: Redis CLI - set and get

Server-side considerations - data storage

Redis - server and CLI

- also manipulate existing values for a given key
  - eg: increment and decrement a value, or simply delete a key
- increment key `notes` value by 1

```
incr notes
```

- decrement key `notes` value by 1

```
decr notes
```

- we can then increment or decrement by a specified amount

```
// increment by 10
incrby notes 10
// decrement by 5
decrby notes 5
```

- delete our key

```
// single key deletion
del notes
// multiple keys deletion
del notes notes2 notes3
```

Image - Client-side and server-side computing

Server-side considerations - data storage

Redis and Node.js setup

- test Redis with our Node.js app
- new test app called `424-node-redis1`

```
|- 424-node-redis1
   |- app
      |- assets
   |- node_modules
   |- package.json
   |- server.js
```

```
Drs-MacBook-Air-2:~ ancientlives$ redis-cli
127.0.0.1:6379> set notes 0
OK
127.0.0.1:6379> get notes
"0"
127.0.0.1:6379> incr notes
(integer) 1
127.0.0.1:6379> incr notes
(integer) 2
127.0.0.1:6379> get notes
"2"
127.0.0.1:6379> decr notes
(integer) 1
127.0.0.1:6379> get notes
"1"
127.0.0.1:6379> incrby notes 10
(integer) 11
127.0.0.1:6379> get notes
"11"
127.0.0.1:6379> decrby notes 5
(integer) 6
127.0.0.1:6379> get notes
"6"
```

Figure 16: Redis CLI - increment and decrement

- create new file, `package.json` to track project
  - eg: dependencies, name, description, version...

Server-side considerations - data storage

```
{
  "name": "424-node-redis1",
  "version": "1.0.0",
  "description": "test app for node and redis",
  "main": "server.js",
  "dependencies": {
    "body-parser": "^1.14.1",
    "express": "^4.13.3",
    "redis": "^2.3.0"
  },
  "author": "ancientlives",
  "license": "ISC"
}
```

Redis and Node.js - package.json

- we can write the `package.json` file ourselves or use the interactive option

```
npm init
```

- then add extra dependencies, eg: Redis, using

```
npm install redis --save
```

- use `package.json` to help with app management and abstraction...

Server-side considerations - data storage

Redis and Node.js - set notes value

- add Redis to our earlier test app
- import and use Redis in the `server.js` file

```
...
var express = require("express"),
    http = require("http"),
    bodyParser = require("body-parser"),
    jsonApp = express(),
    redis = require("redis");
...
```

- create client to connect to Redis from Node.js

```
//create client to connect to Redis
redisConnect = redis.createClient();
```

- then use Redis, for example, to store access total for notes on server

```
redisConnect.incr("notes");
```

- check Redis command line for change in `notes` value

```
get notes
```

Server-side considerations - data storage

Redis and Node.js - get notes value

- now set the counter value for our notes
  - add our counter to the application to record access count for notes
- use the `get` command with Redis to retrieve the incremented values for the `notes` key

```
redisConnect.get("notes", function(error, notesCounter) {
  //set counter to int of value in Redis or start at 0
  notesTotal.notes = parseInt(notesCounter,10) || 0;
});
```

- `get` accepts two parameters - `error` and return value
- Redis stores values and strings
  - convert string to integer using `parseInt()`
  - two parameters - return value and `base-10` value of the specified number
- value is now being stored in a global variable `notesTotal`
  - declared in `server.js`

```
var express = require("express"),
    http = require("http"),
    bodyParser = require("body-parser"),
    jsonApp = express(),
    redis = require("redis"),
    notesTotal = {};
```

Server-side considerations - data storage

Redis and Node.js - get notes value

- store notes counter value in Redis
- create new route in `server.js`
  - monitor the returned JSON for the counter

```
//json get route
jsonApp.get("/notesTotal.json", function(req, res) {
  res.json(notesTotal);
});
```

- start using it with our application
  - load by default, within event handler...
- render to DOM
- store as a internal log record
- link to create note event handler...
- DEMO - 424-node-redis1

Image - Client-side and server-side computing

Figure 17: Node and Redis - 424-node-redis1

Server-side considerations - data storage

MongoDB - intro

- MongoDB is another example of a NoSQL based data store
  - a database that enables us to store our data on disk
- unlike MySQL, for example, it is not in a relational format
- MongoDB is best characterised as a **document-oriented** database
- conceptually may be considered as storing objects in collections
- stores its data using the BSON format
  - consider similar to JSON
  - use JavaScript for working with MongoDB

Server-side considerations - data storage

MongoDB - document oriented

- SQL database, data is stored in tables and rows
- MongoDB, by contrast, uses **collections** and **documents**
- comparison often made between a collection and a table
  - **NB:** a document is quite different from a table
  - a document can contain a lot more data than a table
- a noted concern with this document approach is duplication of data
- one of the trade-offs between NoSQL (MongoDB) and SQL
- SQL - goal of data structuring is to normalise as much as possible
  - thereby avoiding duplicated information
- NoSQL (MongoDB) - provision a data store, as easy as possible for the application to use

---

Server-side considerations - data storage

MongoDB - BSON

- BSON is the format used by MongoDB to store its data
- effectively, JSON stored as binary with a few notable differences
  - eg: `ObjectId` values - data type used in MongoDB to uniquely identify documents
  - created automatically on each document in the database
  - often considered as analogous to a primary key in a SQL database
- `ObjectId` is a large pseudo-random number
- for nearly all practical occurrences, assume number will be unique
- might cease to be unique if server can't keep pace with number generation…
- other interesting aspect of `ObjectId`
  - they are partially based on a timestamp
  - helps us determine when they were created

Server-side considerations - data storage

MongoDB - general hierarchy of data

- in general, MongoDB has a three tiered data hierarchy
- 1. database
  - normally one database per app
  - possible to have multiple per server
  - same basic role as DB in SQL
- 2. collection
  - a grouping of similar pieces of data
  - documents in a collection
  - name is usually a noun
  - resembles in concept a table in SQL
  - documents do not require the same schema
- 3. document
  - a single item in the database
  - data structure of field and value pairs
  - similar to objects in JSON
  - eg: an individual user record

Server-side considerations - data storage

MongoDB - install and setup

- install on Linux
- install on Mac OS X
  - again, we can use **Homebrew** to install MongoDB

```
// update brew packages
brew update
// install MongoDB
brew install mongodb
```

- then follow the above OS X install instructions to set paths…
- install on Windows

Server-side considerations - data storage

MongoDB - a few shell commands

- issue following commands at command line to get started - OS X etc

```
// start MongoDB server - terminal window 1
mongod
// connect to MongoDB - terminal window 2
mongo
```

- switch to, create a new DB (if not available), and drop a current DB as follows

```
// list available databases
show dbs
// switch to specified db
use 424db1
// show current database
db
// drop current database
db.dropDatabase();
```

- DB is not created permanently until data is created and saved
  - insert a record and save to current DB
- only permanent DB is the local `test` DB, until new DBs created…

Server-side considerations - data storage

MongoDB - a few shell commands

- add an initial record to a new `424db1` database.

```
// select/create db
use 424db1
// insert data to collection in current db
db.notes.insert({
...   "travelNotes": [{
...   "created": "2015-10-12T00:00:00Z",
...   "note": "Curral das Freiras..."
...   }]
... })
```

- our new DB `424db1` will now be saved in Mongo
- we've created a new collection, `notes`

```
// show databases
show dbs
// show collections
show collections
```

Server-side considerations - data storage

MongoDB - test app

- now create a new test app for use with MongoDB
- create and setup app as before
  - eg: same setup pattern as Redis test app
- add **Mongoose** to our app
  - use to connect to MongoDB
  - helps us create a schema for working with DB
- update our `package.json` file
  - add dependency for Mongoose

```
// add mongoose to app and save dependency to package.json
npm install mongoose --save
```

- test server and app as usual from app's working directory

```
node server.js
```

Server-side considerations - data storage

MongoDB - Mongoose schema

- use **Mongoose** as a type of bridge between Node.js and MongoDB
- works as a client for MongoDB from Node.js applications
- serves as a useful data modeling tool
  - represent our documents as objects in the application
- a data model
  - object representation of a document collection within data store
  - helps specify required fields for each collection's document
  - known as a schema in Mongoose, eg: `NoteSchema`

```
var NoteSchema = mongoose.Schema({
  "created": Date,
  "note": String
});
```

- using schema, build a model
  - by convention, use first letter uppercase for name of data model object

```
var Note = mongoose.model("Note", NoteSchema);
```

- now start creating objects of this model type using JavaScript

```
var funchalNote = new Note({
"created": "2015-10-12T00:00:00Z",
"note": "Curral das Freiras..."
});
```

- then use the Mongoose object to interact with the MongoDB
  - using functions such as `save` and `find`

Server-side considerations - data storage

MongoDB - test app

- with our new DB setup, our schema created
  - now start to add notes to our DB, `424db1`, in MongoDB
- in our `server.js` file
  - need to connect Mongoose to `424db1` in MongoDB
  - define our schema for our notes
  - then model a note
  - use model to create a note for saving to `424db1`

```
...
//connect to 424db1 DB in MongoDB
mongoose.connect('mongodb://localhost/424db1');
//define Mongoose schema for notes
var NoteSchema = mongoose.Schema({
  "created": Date,
  "note": String
});
//model note
var Note = mongoose.model("Note", NoteSchema);
...
```

Server-side considerations - data storage

MongoDB - test app

- then update app's `post` route to save note to `424db1`

```
//json post route - update for MongoDB
jsonApp.post("/notes", function(req, res) {
  var newNote = new Note({
    "created":req.body.created,
    "note":req.body.note
  });
  newNote.save(function (error, result) {
    if (error !== null) {
      console.log(error);
      res.send("error reported");
    } else {
      Note.find({}, function (error, result) {
        res.json(result);
      })
    }
  });
});
```

Server-side considerations - data storage

MongoDB - test app

- update our app's `get` route for serving these notes

```
//json get route - update for mongo
jsonApp.get("/notes.json", function(req, res) {
  Note.find({}, function (error, notes) {
   //add some error checking...
```

```
    res.json(notes);
  });
});
```

- modify `buildNotes()` function in `json_app.js` to get return correctly

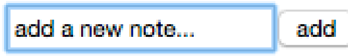
```
...
//get travelNotes
var $travelNotes = response;
...
```

- now able to enter, save, read notes for app
- notes data is stored in the `424db1` database in MongoDB
- notes are loaded from DB on page load
- notes are updated from DB for each new note addition
- DEMO - 424-node-mongo1

Image - Client-side and server-side computing

Node and Mongo

add note



Funchal note...

Monte note...

another note from Curral das Freiras...

on a trip to Porto Moniz...

app's copyright information, additional links...

Figure 18: Node and MongoDB - 424-node-mongo1

Resources

JavaScript

- MDN - Async/Await
- MDN - ES Modules - import
- MDN - ES Modules - export
- MDN - Fetch API
- MDN - Meta Programming
- MDN - Promises
- MDN - Prototype
- MDN - Proxy
- MDN - JS