Comp 324/424 - Client-side Web Design

Spring Semester 2020 - Week 13

Dr Nick Hayward

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
  - *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.*

```javascript
//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*

```javascript
//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

some benefits of overriding the constructor property

```
//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}`);
}

// 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}`);
```

- 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

```
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

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

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', 'buildMeta:release', 'clean']);
```

- 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.*

```json
{
    "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:release', 'clean']);
```

- 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*

```html
<!-- 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

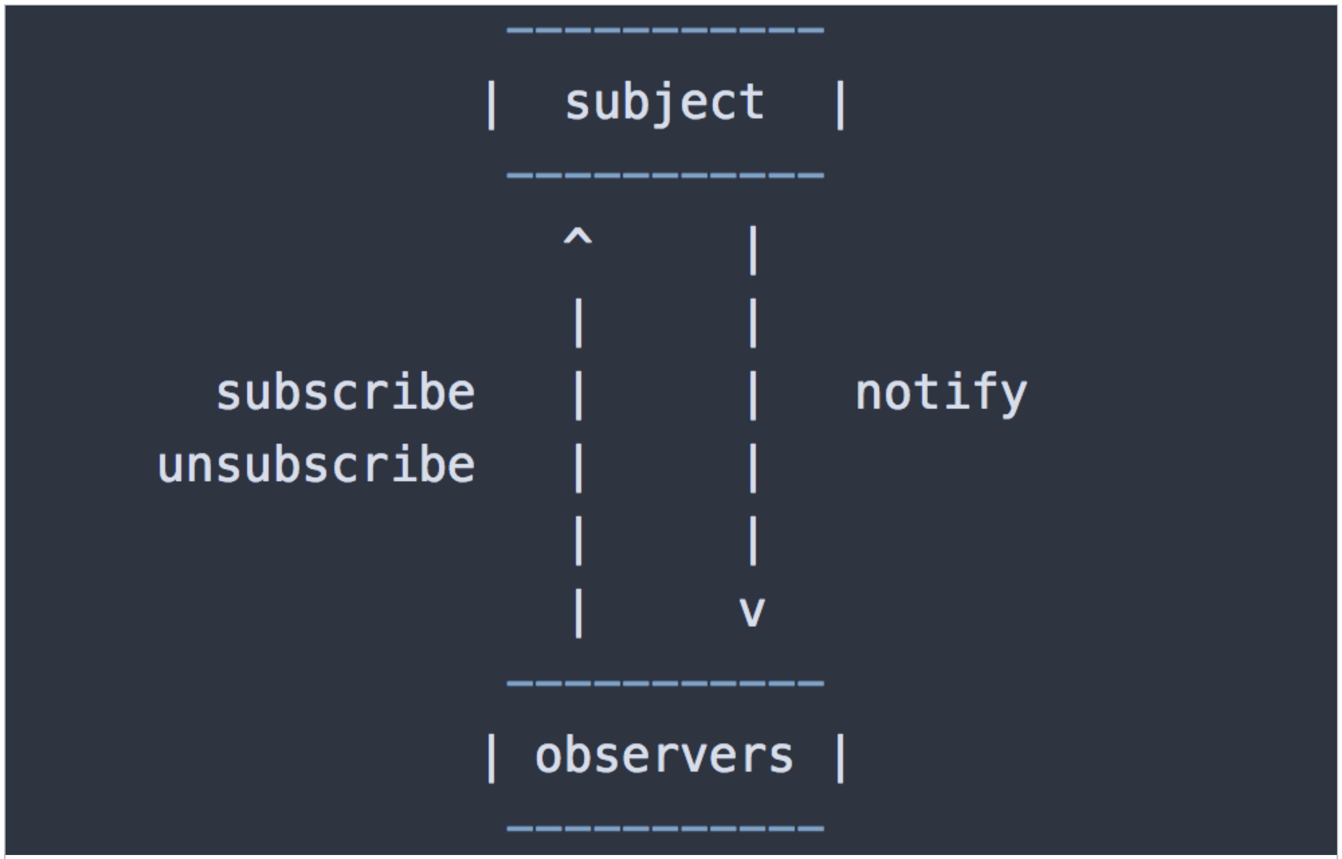
- current `watch` task includes support for CSS, JS, and HTML
- includes checks for modifications
  - *e.g. to any defined `src` directories for CSS and JS*
  - *monitors any HTML files in the app's root directory*
- a working `watch` task is as follows

```
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'],
    },
},
```

Design Patterns - Observer - intro

- *observer* pattern is used to help define a *one to many* dependency between objects
- as **subject** (object) changes state
  - *any dependent **observers** (object/s) are then notified automatically*
  - *and then may update accordingly*
- managing changes in state to keep app in sync
- creating bindings that are event driven
  - *instead of standard push/pull*
- standard usage for this pattern with bindings
  - *one to many*
  - *one way*
  - *commonly event driven*

Image - Observer Pattern



Observer Pattern

Design Patterns - Observer - notifications

- observer pattern creates a model of event subscription with notifications
- benefit of this pattern
  - *tends to promote loose coupling in component design and development*
- pattern is used a lot in JavaScript based applications
  - *user events are a common example of this usage*
- pattern may also be referenced as *Pub/Sub*
  - *there are differences between these patterns - be careful...*

Design Patterns - Observer - Usage

The observer pattern includes two primary objects,

- subject
  - *provides interface for observers to subscribe and unsubscribe*
  - *sends notifications to observers for changes in state*
  - *maintains record of subscribed observers*
  - *e.g. a click in the UI*
- observer
  - *includes a function to respond to subject notifications*
  - *e.g. a handler for the click*

Design Patterns - Observer - Example

```javascript
// constructor for subject
function Subject () {
  // keep track of observers
  this.observers = [];
}

// add subscribe to constructor prototype
Subject.prototype.subscribe = function(fn) {
  this.observers.push(fn);
};

// add unsubscribe to constructor prototype
Subject.prototype.unsubscribe = function(fn) {
  // ...
};

// add broadcast to constructor prototype
Subject.prototype.broadcast = function(status) {
  // each subscriber function called in response to state change...
  this.observers.forEach((subscriber) => subscriber(status));
};

// instantiate subject object
const domSubject = new Subject();

// subscribe & define function to call when broadcast message is sent
domSubject.subscribe((status) => {
  // check dom load
  let domCheck = status === true ? `dom loaded = ${status}` : `dom still loading...`;
  // log dom check
  console.log(domCheck)
});

document.addEventListener('DOMContentLoaded', () => domSubject.broadcast(true));
```

Design Patterns - Observer - Example

- Demo - Observer - Broadcast, Subscribe, & Unsubscribe

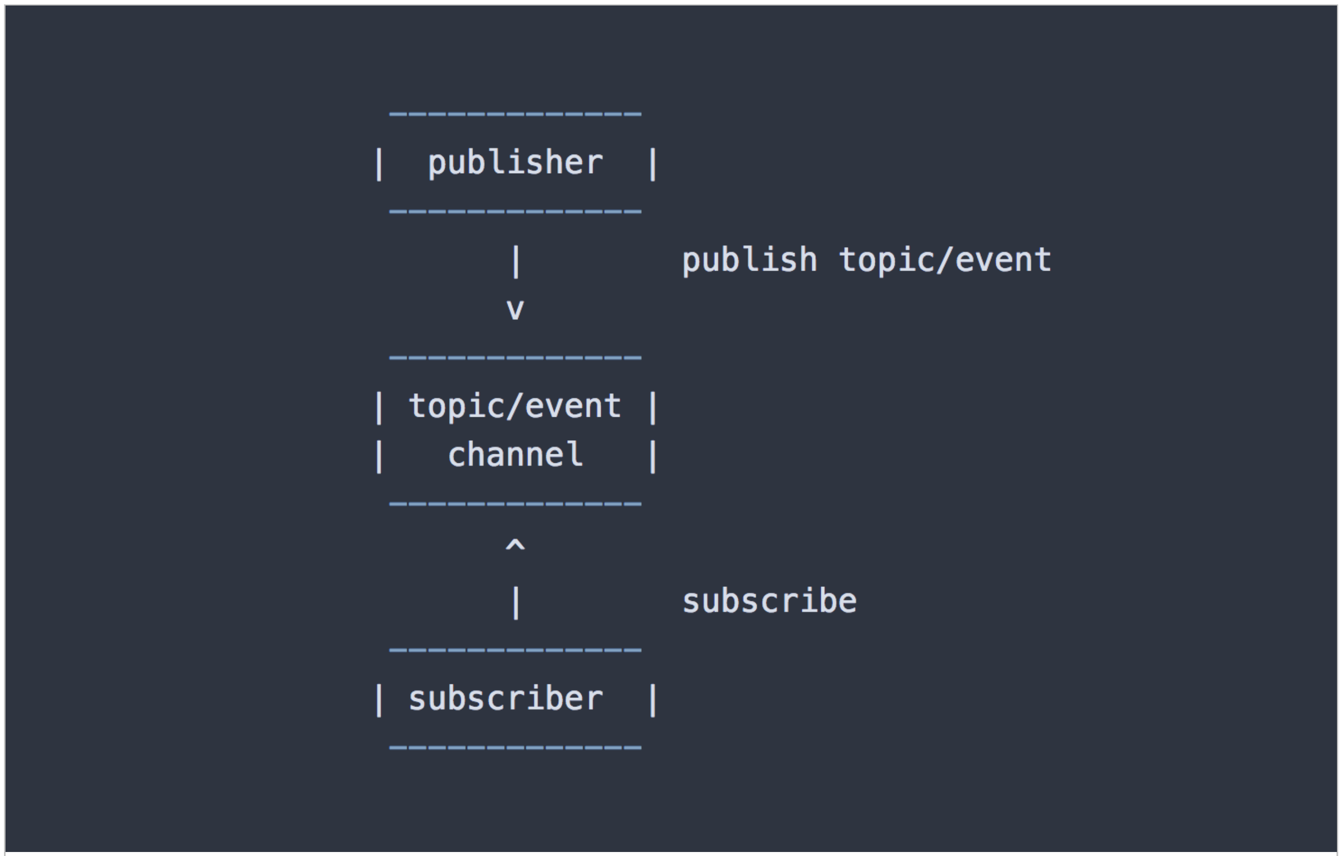
Design Patterns - Pub/Sub - intro

- variation of standard *observer* pattern is *publication and subscription*
  - *commonly known as PubSub pattern*
- popular usage in JavaScript
- *PubSub* pattern publishes a *topic* or *event* channel
- publication acts as a *mediator* or *event system* between
  - *subscriber objects wishing to receive notifications*
  - *and publisher object announcing an event*
- easy to define specific events with *event system*
- events may then pass custom arguments to a subscriber
- trying to avoid potential dependencies between objects
  - *subscriber objects and the publisher object*

Design Patterns - Pub/Sub - abstraction

- inherent to this pattern is the simple abstraction of responsibility
- publishers are unaware of nature or type of subscribers for messages
- subscribers are unaware of the specifics for a given publisher
- subscribers simply identify their interest in a given topic or event
  - *then receive notifications of updates for a given subscribed channel*
- primary difference with *observer* pattern
  - *PubSub abstracts the role of the subscriber*
- *subscriber* simply needs to handle data broadcasts by a *publisher*
- creating an abstracted *event system* between objects
  - *abstraction of concerns between publisher and subscriber*

Image - Publish/Subscribe Pattern



```
 -------------
|  publisher  |
 -------------
       |        publish topic/event
       v
 -------------
| topic/event |
|   channel   |
 -------------
       ^
       |        subscribe
 -------------
| subscriber  |
 -------------
```

PubSub Pattern

Design Patterns - Pub/Sub - benefits

- *observer* and *PubSub* patterns help developers
  - *better understanding of relationships within an app’s logic and structure*
- need to identify aspects of our app that contain direct relationships
- many direct relationships may be replaced with patterns
  - *subjects and observers*
  - *publishers and observers*
- tightly coupled code can quickly create issues
  - *maintenance, scale, modification, clarity of code and logic…*
  - *semmingly minor changes may often create a cascade or waterfall effect in code*
- a known side effect of tightly couple code
  - *frequent need to mock usage &c. in testing*
  - *time consuming and error prone as app scales…*
- *PubSub* helps create smaller, loosely coupled blocks
  - *helps improve management of an app*
  - *promotes code reuse*

Design Patterns - Pub/Sub - basic example - part 1 - event system

```
// constructor for pubsub object
function PubSub () {
  this.pubsub = {};
}

// publish - expects topic/event & data to send
PubSub.prototype.publish = function (topic, data) {
  // check topic exists
  if (!this.pubsub[topic]){
    console.log(`publish - no topic...`);
    return false;
  }
  // loop through pubsub for specified topic - call subscriber functions...
  this.pubsub[topic].forEach(function(subscriber) {
      subscriber(data || {});
    });
};

// subscribe - expects topic/event & function to call for publish notification
PubSub.prototype.subscribe = function (topic, fn) {
  // check topic exists
  if (!this.pubsub[topic]) {
    // create topic
    this.pubsub[topic] = [];
    console.log(`pubsub topic initialised...`);
  }
  else {
    // log output for existing topic match
    console.log(`topic already initialised...`);
  }
  // push subscriber function to specified topic
  this.pubsub[topic].push(fn);
};
```

Design Patterns - Pub/Sub - basic example - part 2 - usage

```
// basic log output
var logger = data => { console.log( `logged: ${data}` ); };

// test function for subscriber
var domUpdater = function (data) {
  document.getElementById('output').innerHTML = data;
}

// instantiate object for PubSub
const pubSub = new PubSub();

// subscriber tests
pubSub.subscribe( 'test_topic', logger );
pubSub.subscribe( 'test_topic2', domUpdater );
pubSub.subscribe( 'test_topic', logger );

// publisher tests
pubSub.publish('test_topic', 'hello subscribers of test topic...');
pubSub.publish('test_topic2', 'update notification for test topic2...');
```

- Demo - Pub/Sub

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*

```javascript
// 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*

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

```
// 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

```
// 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.

```javascript
// 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.

```javascript
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*

```javascript
// 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

```javascript
// 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;
```

Demos

- Design Patterns
  - *Observer - Broadcast, Subscribe, & Unsubscribe*
  - *Pub/Sub*
- JavaScript - Prototype
  - *basic prototype*
  - *basic set prototype*
  - *basic prototype object*
  - *basic prototype object properties*
  - *basic prototype dynamic*
  - *basic constructor check*
  - *inheritance with prototype*
  - *inheritance with prototype - updated*
  - *configure object properties*
- JavaScript - ES Class
  - *basic ES Class*
  - *basic Prototype equivalent*

Resources

- Design Patterns
  - *Observer - Wikipedia*
  - *Pub/Sub Messaging - AWS*
  - *Pub/Sub - Wikipedia*
- JavaScript - Prototype
  - *MDN - Object Prototypes*
  - *MDN - Inheritance and the prototype chain*
- JavaScript - ES Class
  - *MDN - Classes*
- JavaScript - Proxy
  - *MDN - Proxy*
  - *MDN - Meta Programming*
- Project tools
  - *Grunt JavaScript Task Runner*
  - *Webpack Asset Bundler*