Comp 324/424 - Client-side Web Design

Fall Semester 2019 - Week 13

Dr Nick Hayward

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
```

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

Server-side considerations - data storage

Redis and Node.js - package.json

```
{
  "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"
}
```

- 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

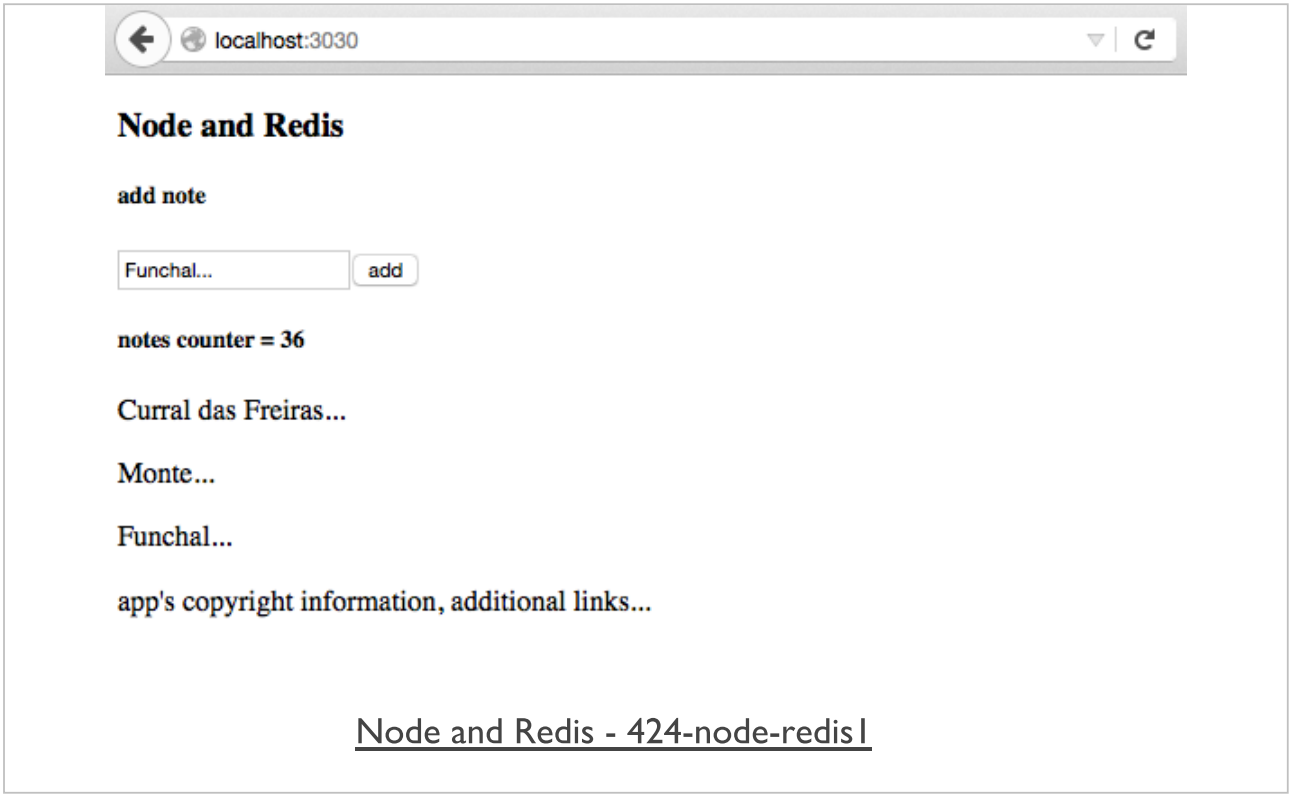
Redis and Node.js - get notes value

- 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



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

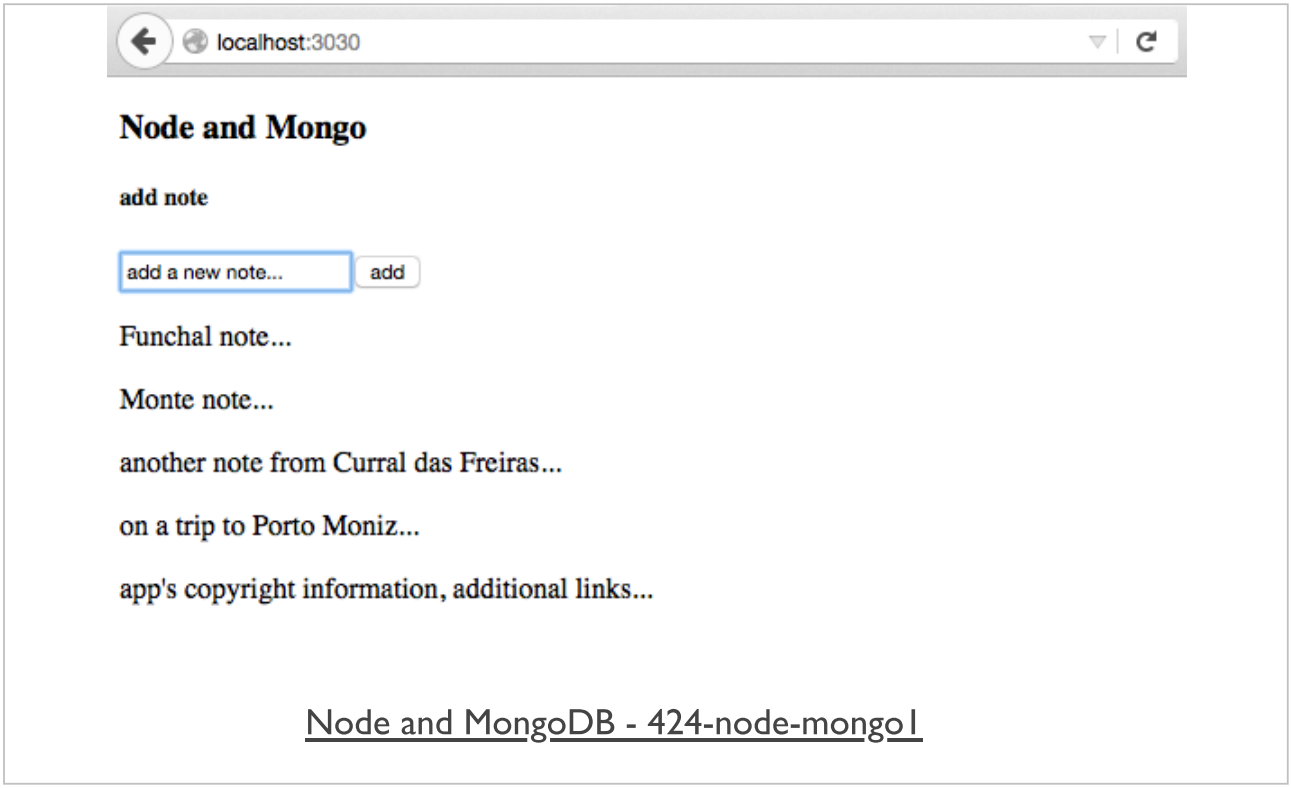
```
//json get route - update for mongo
jsonApp.get("/notes.json", function(req, res) {
  Note.find({}, function (error, notes) {
   //add some error checking...
   res.json(notes);
  });
});
```

- 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 MongoDB - 424-node-mongo1

Client-side - Data - Node, Express, MongoDB &c.

---

extra notes

- Heroku
  - *Heroku & Git*
  - *Heroku & MongoDB*
  - *Heroku & Postman*
- Node.js
  - *Node.js outline*
  - *Node.js updating*
- Node.js & Express
  - *Node.js and Express*
  - *Node.js & Express starter*
- Node.js, Express, and MongoDB
  - *Node.js and MongoDB*
- Node.js API
  - *Data stores & APIs - MongoDB and native driver*
  - *Node Todos API*
  - *Testing - Node Todos API*
- Node.js & Web Sockets
  - *Node.js & Socket.io*

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 lit
```

- `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.o
// 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
try {
// updates to original prototype are overridden - error is returned for second in
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 I

- *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...Y
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

Resources

- JavaScript - Prototype
  - *MDN - Object Prototypes*
  - *MDN - Inheritance and the prototype chain*
- MongoDB
  - *MongoDB - For Giant Ideas*
  - *MongoDB - Getting Started (Node.js driver edition)*
  - *MongoDB - Getting Started (shell edition)*
- Mongoose
  - *MongooseJS Docs*
- Node.js
  - *Node.js home*
  - *Node.js - download*
  - *ExpressJS*
  - *ExpressJS body-parser*