Comp 322/422 - Software Development for Wireless and Mobile Devices

Fall Semester 2019 - Week 6

Dr Nick Hayward

Mobile Design & Development - Data Usage and Persistency

Fun Exercise

Four apps, one per group

- Book Exchange Map - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/books/
- Chat Map - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/chat/
- Cycle Map - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/cycle/
- Physio Map - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/physio/

For your assigned app, consider the following

- relevant use of mapping and geolocation within the app
  - *does the map &c. help the app?*
  - *what is the value of geolocation in the app?*
- what type of data needs to stored in this app?
  - *local options...*
  - *remote or cloud options...*

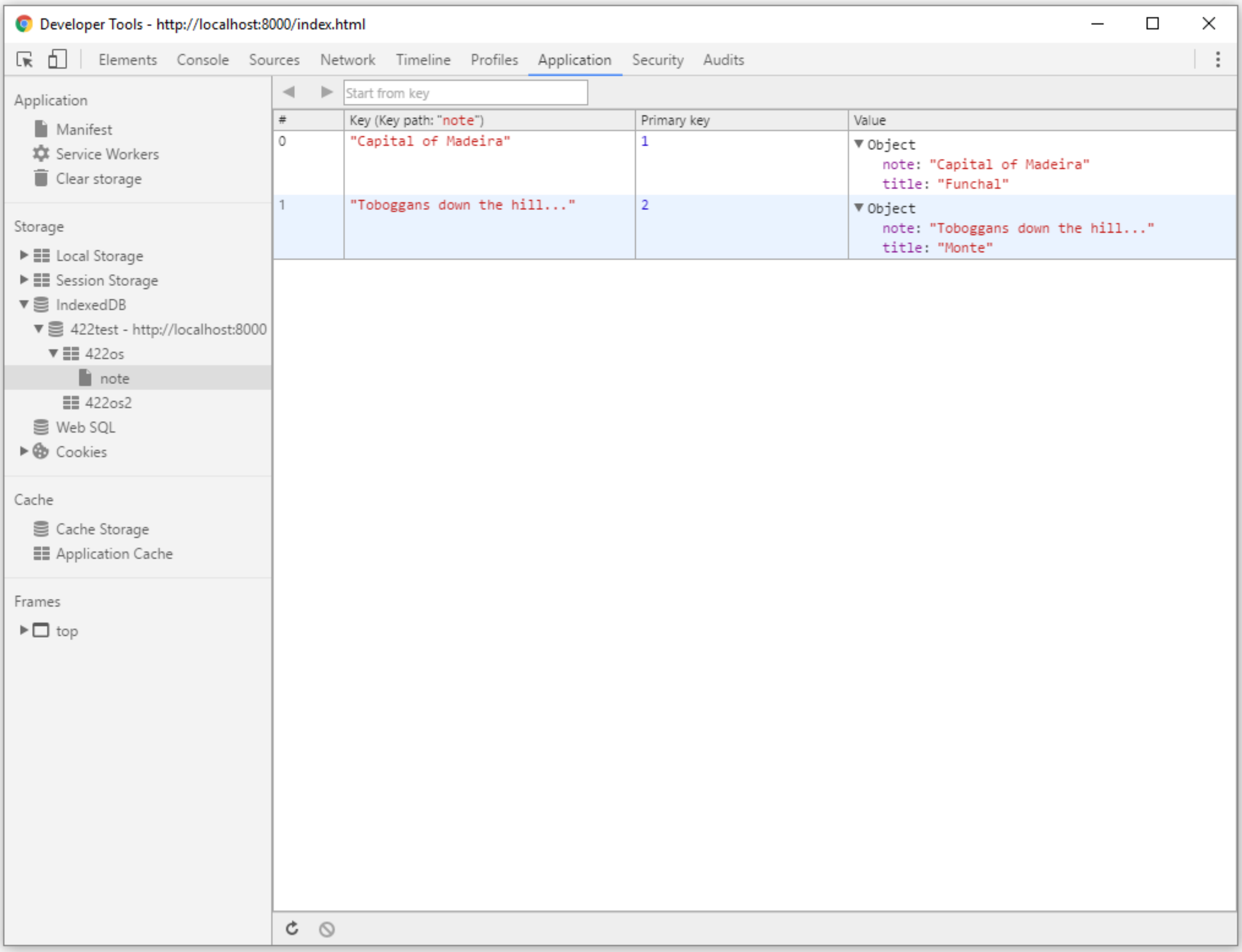
~ 10 minutes

Cordova app - IndexedDB - Recap

Material covered so far:

- general intro
- checked IndexedDB availability as part of `deviceready` event
  - *created reference for later use...*
- general usage
  - *connection &c.*
- event listeners
  - *`success`, `error`, `upgradeneeded`, `blocked`*
- create a new DB
  - *check persistence*
  - *work with success and fail callbacks*
- object stores
- add data
- work with data handlers
- multiple object stores, notes...
- keys
- ...

Image - IndexedDB Support



#	Key (Key path: "note")	Primary key	Value
0	"Capital of Madeira"	1	▼Object note: "Capital of Madeira" title: "Funchal"
1	"Toboggans down the hill..."	2	▼Object note: "Toboggans down the hill..." title: "Monte"

DataTest2 - test IndexedDB - unique keys 2

Cordova app - IndexedDB - data test 2

database - part 16 - read data

- now able to save our notes to the IndexedDB
- need to read this data, and then load it into our application
- use the same underlying pattern for read and write
  - *use a transaction, and the request will be asynchronous*
  - *modify our transaction for `readonly`*

```
// create transaction
var dbTransaction2 = db.transaction(["422os"],"readonly");
```

- then use our new transaction get the required object store,

```
// define data object store
var dataStore2 = dbTransaction.objectStore("422os");
```

- then request our value from the database,

```
// request value - key &c.
var object1 = dataStore2.get(key);
```

- then use returned value for rendering...

Cordova app - IndexedDB - data test 2

database - part 17 - read data

- update our HTML with a button to load and test our data from IndexedDB,

```
...
<input type="button" id="loadNote" data-icon="refresh" value="Load Note" data-inline="true
...
```

- add our event handler for the button
  - *allows us to call the* `loadNoteData()` *function for querying the IndexedDB*

```
// handler for load note button
$("#loadNote").on("tap", function(e) {
  e.preventDefault();
  // get requested data for specified key
  loadNoteData(1);
});
```

Cordova app - IndexedDB - data test 2

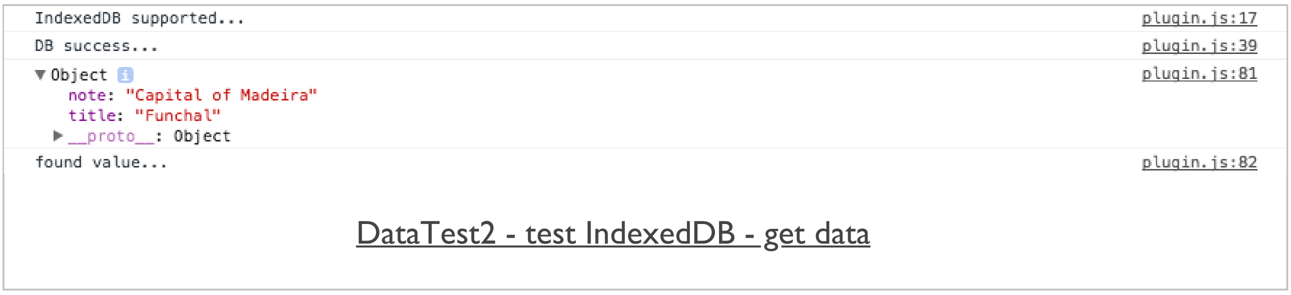
database - part 18 - read data

- need to add our new function to load the data from the object store

```
function loadNoteData(key) {
 var dbTransaction = db.transaction(["422os"],"readonly");
 // define data object store
 var dataStore2 = dbTransaction.objectStore("422os");
 // request value - use defined key
 var object1 = dataStore2.get(key);
 // do something with return
 object1.onsuccess = function(e) {
   var result = e.target.result;
   //output to console for testing
   console.dir(result);
   console.log("found value...");
 }
}
```

- use transaction to create connection to specified object store in IndexedDB
- able to request a defined value using a specified key
  - *in this example `key` 1 for the object store `422os`*
- process return value for use in application

Image - IndexedDB Support



```
IndexedDB supported...                                   plugin.js:17
DB success...                                            plugin.js:39
▼Object i                                                plugin.js:81
    note: "Capital of Madeira"
    title: "Funchal"
  ▶ __proto__: Object
found value...                                           plugin.js:82
```

DataTest2 - test IndexedDB - get data

Cordova app - IndexedDB - data test 2

database - part 19 - read more data

- retrieving a single, specific value for a given key is obviously useful
    - *may become limited in practical application usage*
- IndexedDB provides an option to retrieve multiple data values
- uses an option called a `cursor`
    - *helps us iterate through specified data within our IndexedDB*
- use these cursors to create iterators with optional filters
    - *using range within a specified dataset*
    - *also add a required direction*
- creating and working with a cursor requires
    - *a transaction*
    - *performs an asynchronous request*

Cordova app - IndexedDB - data test 2

database - part 19 - read more data

- create our transaction,

```
var dbTransaction = db.transaction(["422os"],"readonly");
```

- retrieve our object store containing the required data

```
// define data object store
var dataStore3 = dbTransaction.objectStore("422os");
```

- now create our `cursor` for use with the required object store,

```
var cursor = dataStore3.openCursor();
```

- with this connection to the required object store in our specified IndexedDB
  - *now process the return values for our request*

Cordova app - IndexedDB - data test 2

database - part 20 - read more data

- use cursor to iterate through return results
  - *work with specified object store within our standard `success` handler*

```
cursor.onsuccess = function(e) {
  var result = e.target.result;
  if (result) {
    console.dir("notes", result.value);
    console.log("notes", result.key);
    result.continue();
 }
}
```

- new success handler is working with a passed object for the result from our IndexedDB
- object, `402result`, contains
  - *required keys, data, and a method to iterate through the returned data*
- `continue()` method is the iterator for this cursor
  - *allows us to iterate through our specified object store*

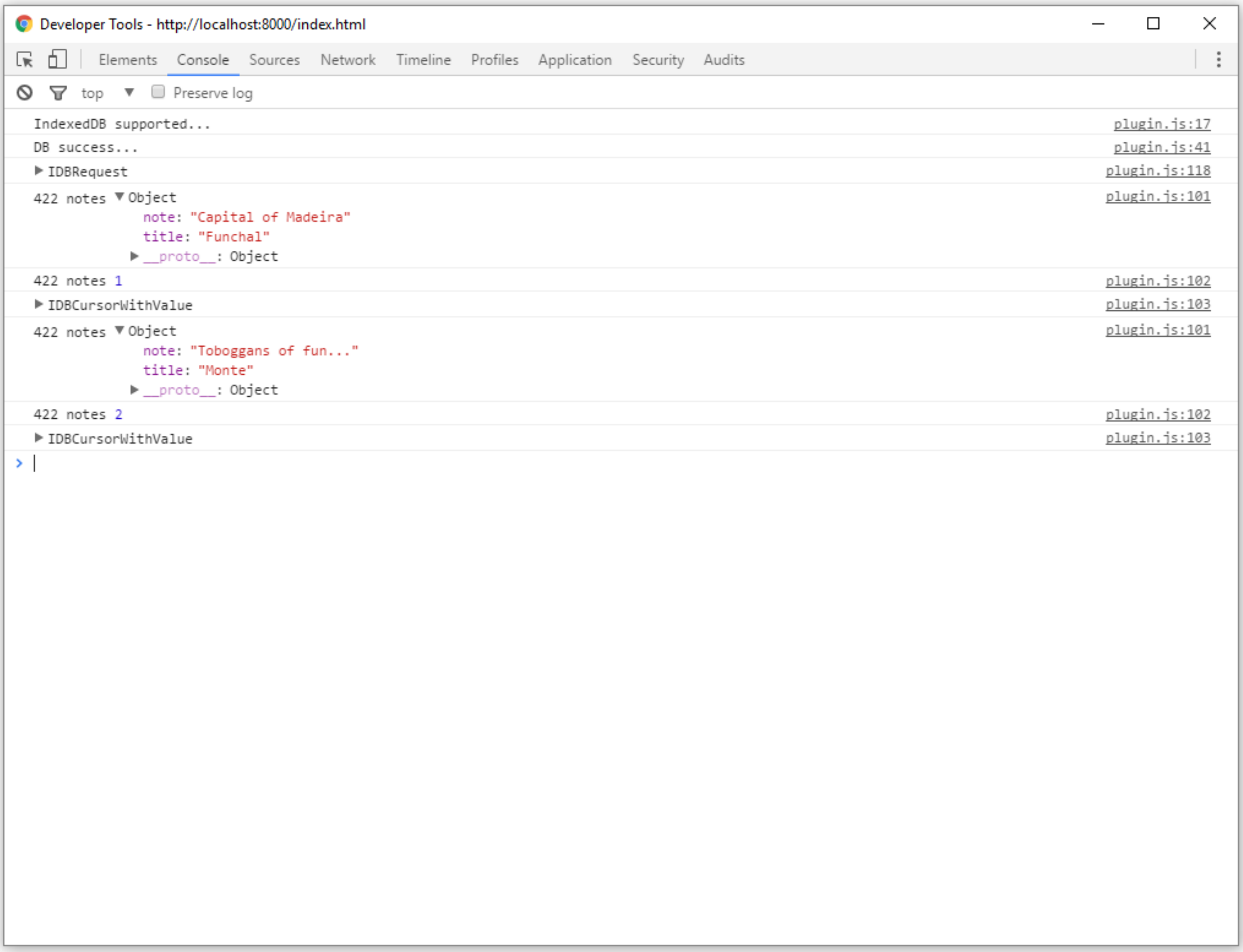
Cordova app - IndexedDB - data test 2

database - part 21 - read more data

- add an option to view all of the notes within our IndexedDB
- using the following new function, `loadNotes()`

```
function loadNotes() {
  // create transaction
  var dbTransaction = db.transaction(["422os"],"readonly");
  // define data object store
  var dataStore3 = dbTransaction.objectStore("422os");
  var cursor = dataStore3.openCursor();
  // do something with return...
  cursor.onsuccess = function(e) {
    var result = e.target.result;
    if (result) {
      console.log("422 notes", result.value);
      console.log("422 notes", result.key);
      console.dir(result);
      result.continue();
    }
  }
}
```

Image - IndexedDB Support



```
IndexedDB supported...                                   plugin.js:17
DB success...                                            plugin.js:41
▶ IDBRequest                                             plugin.js:118
422 notes ▼Object                                        plugin.js:101
              note: "Capital of Madeira"
              title: "Funchal"
            ▶ __proto__: Object
422 notes 1                                              plugin.js:102
▶ IDBCursorWithValue                                     plugin.js:103
422 notes ▼Object                                        plugin.js:101
              note: "Toboggans of fun..."
              title: "Monte"
            ▶ __proto__: Object
422 notes 2                                              plugin.js:102
▶ IDBCursorWithValue                                     plugin.js:103
>
```

DataTest2 - test IndexedDB - read more data

Cordova app - IndexedDB - data test 2

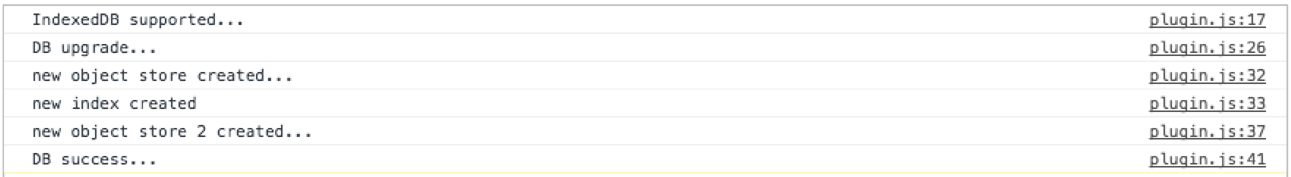
database - part 22 - index

- a primary benefit of using IndexedDB
  - *its support for indexes*
  - *retrieve data from these object stores using the data value itself*
  - *in addition to the standard key search*
- start by adding this option to our object stores
- create an index by using our pattern for an `upgrade` event
  - *creating the index at the same time as the object store*

```
var dataStore = db.createObjectStore("422os", { autoIncrement:true});
// set name of index
dataStore.createIndex("note","note", {unique:false});
```

- creating our object store, `422os`
  - *then using object store result to create and index using `createIndex()`*
  - *first argument for this method is the name for our index*
  - *second is the actual property we want indexing within the object store*
  - *add a set of options, eg: unique or not*
- IndexedDB will then create an index for this object store

Image - IndexedDB Support



```
IndexedDB supported...                                                                                          plugin.js:17
DB upgrade...                                                                                                   plugin.js:26
new object store created...                                                                                     plugin.js:32
new index created                                                                                               plugin.js:33
new object store 2 created...                                                                                   plugin.js:37
DB success...                                                                                                   plugin.js:41
```

DataTest2 - test IndexedDB - create index

Cordova app - IndexedDB - data test 2

database - part 22 - index

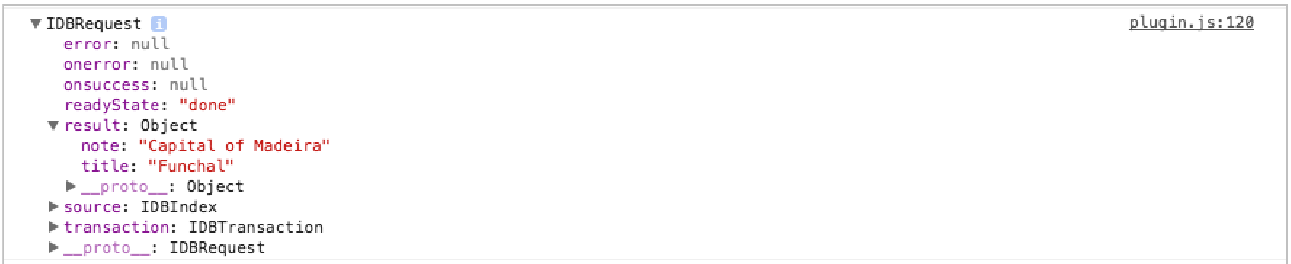
- new index now created
  - *start to add options for querying the database's values*
- need to specify a required index from the applicable object store
- use a transaction to retrieve a given object store
  - *then able to specify required index from that object store*

```
// create transaction
var dbTransaction = db.transaction(["422os"],"readonly");
// define data object store
var dataStore = dbTransaction.objectStore("422os");
// define index
var dataIndex = dataStore.index("note");
```

- we can then request some values using a standard `get` method with this index

```
var note = "Capital of Madeira";
var getRequest = dataIndex.get(note);
```

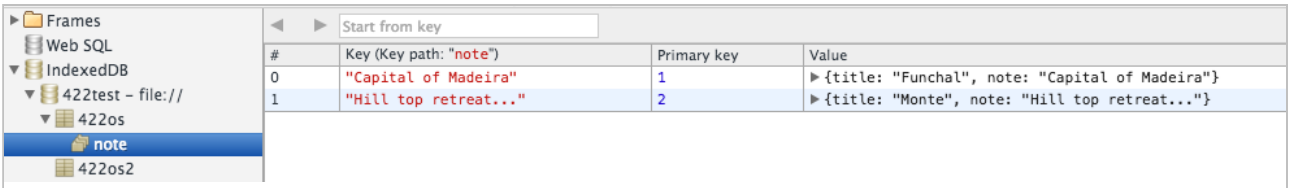
Image - IndexedDB Support



```
▼ IDBRequest ℹ                                                    plugin.js:120
    error: null
    onerror: null
    onsuccess: null
    readyState: "done"
  ▼ result: Object
      note: "Capital of Madeira"
      title: "Funchal"
    ▶ __proto__: Object
  ▶ source: IDBIndex
  ▶ transaction: IDBTransaction
  ▶ __proto__: IDBRequest
```

DataTest2 - test IndexedDB - query index

Image - IndexedDB Support



#	Key (Key path: "note")	Primary key	Value
0	"Capital of Madeira"	1	▶ {title: "Funchal", note: "Capital of Madeira"}
1	"Hill top retreat..."	2	▶ {title: "Monte", note: "Hill top retreat..."}

DataTest2 - test IndexedDB - current index

Cordova app - IndexedDB - data test 2

database - part 23 - index

- we will need to consider queries against an index in much broader terms
- we need to consider the use and application of ranges relative to our index
- use of ranges returns a limited set of data from our object store
- IndexedDB helps us create few different options for ranges
  - ***everything above...***, ***everything below...***, ***something between...***, ***exact***
  - *set ranges either inclusive or exclusive*
  - *request ascending and descending ranges for our results*
- an example range might be limiting a query to a specific word, title, or other key value...

```
// Only match "Madeira"
var singleRange = IDBKeyRange.only("Madeira");
```

- by default, IndexedDB supports the following types of queries
  - *`IDBKeyRange.only()` - Exact match*
  - *`IDBKeyRange.upperBound()` – objects = property below certain value*
  - *`IDBKeyRange.lowerBound()` – objects = property above certain value*
  - *`IDBKeyRange.bound()` – objects = property between certain values*

Server-side considerations - data storage

SQL or NoSQL

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

Server-side considerations - data storage

Redis - intro

- Redis provides an excellent example of NoSQL based data storage
- designed for fast access to frequently requested data
- improvement in performance often due to a reduction in perceived reliability
  - *due to in-memory storage instead of writing to a disk*
- able to flush data to disk
  - *performs this task at given points during uptime*
  - *for majority of cases considered an in-memory data store*
- stores this data in a **key-value** format
  - *similar in nature to standard object properties in JavaScript*
- Redis often a natural extension of conventional data structures
- Redis is a good option for quick access to data
  - *optionally caching temporary data for frequent access*

Server-side considerations - data storage

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

Firebase - mobile platform - what is it?

- other data store and management options now available to us as developers
- depending upon app requirements consider
  - *Firebase*
  - *RethinkDB*
- as a data store, Firebase offers a hosted NoSQL database
  - *data store is JSON-based*
  - *offering quick, easy development from webview to data store*
- syncs an app's data across multiple connected devices in milliseconds
  - *available for offline usage as well*
- provides an API for accessing these JSON data stores
  - *real-time for all connected users*
- Firebase as a hosted option more than just data stores and real-time API access
- Firebase has grown a lot over the last year
  - *many new features announced at Google I/O conference in May 2016*
  - *analytics, cloud-based messaging, app authentication*
  - *file storage, test options for Android*
  - *notifications, adverts...*

Server-side considerations - data storage

working with mobile cross-platform designs

- how can we use Redis, MongoDB, and other data store technologies with Cordova?
- considerations for a multi-platform structure
  - *data*
  - *models*
  - *views*
- authentication
  - *user login*
  - *accounts*
  - *data*

Data considerations in mobile apps

- worked our way through Cordova's File plugin
  - *tested local read and write for files*
- test JS requests with JSON
  - *local and remote files*
  - *remote services and APIs*
- work natively with JS objects
  - *webview*
  - *controller*
  - *local or remote data store or service*

Cross-platform JS - ES6 Generators & Promises - intro

- generators and promises are new to plain JavaScript
  - *introduced with ES6 (ES2015)*
- **Generators** are a special type of function
  - *produce multiple values per request*
  - *suspend execution between these requests*
- *generators* are useful to help simplify convoluted loops
- suspend and resume code execution, &c.
  - *helps write simple, elegant async code*
- **Promises** are a new, built-in object
  - *help development of async code*
- promise becomes a placeholder for a value not currently available
  - *but one that will be available later*

Cross-platform JS - ES6 Generators & Promises - async code and execution

- JS relies on a single-threaded execution model
- query a remote server using standard code execution
  - *block the UI until a response is received and various operations completed*
- we may modify our code to use callbacks
  - *invoked as a task completes*
  - *should help resolve blocking the UI*
- callbacks can quickly create a *spaghetti* mess of code, error handling, logic...
- *Generators* and *Promises*
  - *elegant solution to this mess and proliferation of code*

Cross-platform JS - ES6 Generators & Promises - promises - intro

- a *promise* is similar to a placeholder for a value we currently do not have
  - *but we would like later...*
- it's a guarantee of sorts
  - *eventually receive a result to an asynchronous request, computation, &c.*
- a result will be returned
  - *either a value or an error*
- we commonly use *promises* to fetch data from a server
  - *fetch local and remote data*
  - *fetch data from APIs*

Cross-platform JS - ES6 Generators & Promises - promises - example

```
// use built-in Promise constructor - pass callback function with two parameters (resolve
const testPromise = new Promise((resolve, reject) => {
  resolve("test return");
  // reject("an error has occurred trying to resolve this promise...");
});

// use `then` method on promise - pass two callbacks for success and failure
testPromise.then(data => {
  // output value for promise success
  console.log("promise value = "+data);
}, err => {
  // output message for promise failure
  console.log("an error has been encountered...");
});
```

- use the built-in *Promise* constructor to create a new promise object
- then pass a function
  - *a standard arrow function in the above example*

Cross-platform JS - ES6 Generators & Promises - promises - executor

- function for a Promise is commonly known as an *executor* function
  - *includes two parameters, `resolve` and `reject`*
- *executor* function is called immediately
  - *as the Promise object is being constructed*
- `resolve` argument is called manually
  - *when we need the promise to resolve successfully*
- second argument, `reject`, will be called if an error occurs
- uses the *promise* by calling the built-in `then` method
  - *available on the promise object*
- `then` method accepts two callback functions
  - *`success` and `failure`*
- `success` is called if the *promise* resolves successfully
- the `failure` callback is available if there is an error

Cross-platform JS - ES6 Generators & Promises - promises - example

explicit use of `resolve`

```
/*
* promise1.js
* wrap Array in Promise using resolve()...
*/

let testArray = Promise.resolve(['one', 'two', 'three']);

testArray.then(value => {
console.log(value[0]);
// remove first item from array
value.shift();
// pass value to chained `then`
return value;
})
.then(value => console.log(value[0]));
```

- Demo - Promise.resolve

Cross-platform JS - ES6 Generators & Promises - promises - callbacks & async

- async code is useful to prevent execution blocking
  - *potential delays in the browser*
  - *e.g. as we execute long-running tasks*
- issue is often solved using *callbacks*
  - *i.e. provide a callback that's invoked when the task is completed*
- such long running tasks may result in errors
- issue with callbacks
  - *e.g. we can't use built-in constructs such as `try-catch` statements*

Cross-platform JS - ES6 Generators & Promises - promises - callbacks & async - example

```
try {
  getJSON("data.json", function() {
    // handle return results...
  });
} catch (e) {
  // handle errors...
}
```

- this won't work as expected due to the code executing the callback
  - *not usually executed in the same step of the event loop*
  - *may not be in sync with the code running the long task*
- errors will usually get lost as part of this long running task
- another issue with callbacks is nesting
- a third issue is trying to run parallel callbacks
- performing a number of parallel steps becomes inherently tricky and error prone

Cross-platform JS - ES6 Generators & Promises - promises - further details

- a *promise* starts in a pending state
  - *we know nothing about the return value*
  - *promise is often known as an unresolved promise*
- during execution
  - *if the promise's resolve function is called*
  - *the promise will move into its fulfilled state*
  - *the return value is now available*
- if there is an error or *reject* method is explicitly called
  - *the promise will simply move into a rejected state*
  - *return value is no longer available*
  - *an error now becomes available*
- either of these states
  - *the promise can now no longer switch state*
  - *i.e from rejected to fulfilled and vice-versa...*

Cross-platform JS - ES6 Generators & Promises - promises - concept example

an example of working with a promise may be as follows

- code starts (execution is ready)
- promise is now executed and starts to run
- promise object is created
- promise continues until it resolves
  - *successful return, artificial timeout &c.*
- code for the current promise is now at an end
- promise is now resolved
  - *value is available in the promise*
- then work with resolved promise and value
  - *call `then` method on promise and returned value...*
  - *this callback is scheduled for successful resolve of the promise*
  - *this callback will always be asynchronous regardless of state of promise...*

Cross-platform JS - ES6 Generators & Promises - promises - callbacks & async - example

promise from scratch

```
/*
* promisefromscratch-delay.js
* create a Promise object from scratch...use delay to check usage
* promise may only be called once per execution due to delay and timeout...
*/

// check promise usage relative to timer...either timeout will cause the Promise to call a
function resolveWithDelay(delay) {
  return new Promise(function(resolve, reject) {
    // log Promise creation...
    console.log('promise created...waiting');
        // resolve promise if delay value is less than 3000
    setTimeout(function() {
      resolve(`promise resolved in ${delay} ms`);
    }, delay);
        // resolve promise if delay is greater than 3000
    setTimeout(function() {
      resolve(`promise resolved in 3000ms`);
    }, 3000);
  })
}

// fulfilled with delay of 2000 ms
resolveWithDelay(2000).then(function(value) {
  console.log(value);
});
// fulfilled with default timeout of 3000 ms
// resolveWithDelay(6000).then(function(value) {
//   console.log(value);
// });
```

- Demo - Promise from scratch

Cross-platform JS - ES6 Generators & Promises - promises - explicitly reject

- two standard ways to reject a promise
- e.g. explicit rejection of promise

```
const promise = new Promise((resolve, reject) => {
    reject("explicit rejection of promise");
});
```

- once the promise has been rejected
  - *an error callback will always be invoked*
  - *e.g. through the calling of the `then` method*

```
promise.then(
  () => fail("won't be called..."),
  error => pass("promise was explicitly rejected...");
);
```

- also chain a `catch` method to the `then` method
- as an alternative to the error callback. e.g.

```
promise.then(
  () => fail("won't be called..."))
  .catch(error => pass("promise was explicitly rejected..."));
```

Cross-platform JS - ES6 Generators & Promises - promises - example

promise error handling

```
/*
* promise-basic-error1.js
* basic example usage of promise error handling and order...
*/

Promise
  .resolve(1)
  .then(x => {
    if (x === 2) {
      console.log('val resolved as', x);
    } else {
      throw new Error('test failed...')
    }
  })
  .catch(err => console.error(err));
```

- Demo - Promise error handling with catch

Cross-platform JS - ES6 Generators & Promises - promises - real-world promise - getJSON

```
// create a custom get json function
function getJSON(url) {
  // create and return a new promise
  return new Promise((resolve, reject) => {
    // create the required XMLHttpRequest object
    const request = new XMLHttpRequest();
    // initialise this new request - open
    request.open("GET", url);
    // register onload handler - called if server responds
    request.onload = function() {
      try {
        // make sure response is OK - server needs to return status 200 code...
        if (this.status === 200) {
          // try to parse json string - if success, resolve promise successfully with valu
          resolve(JSON.parse(this.response));
        } else {
          // different status code, exception parsing JSON &c. - reject the promise...
          reject(this.status + " " + this.statusText);
        }
      } catch(e) {
        reject(e.message);
      }
    };

    // if error with server communication - reject the promise...
    request.onerror = function() {
      reject(this.status + " " + this.statusText);
    };

    // send the constructed request to get the JSON
    request.send();
  });
}
```

Cross-platform JS - ES6 Generators & Promises - promises - real-world promise - usage

```
// call getJSON with required URL, then method for resolve object, and catch for error
getJSON("test.json").then(response => {
  // check return value from promise...
  response !== null ? "response obtained" : "no response";
}).catch((err) => {
  // Handle any error that occurred in any of the previous promises in the chain.
    console.log('error found = ', err); // not much to show due to return of jsonp from fl
});
```

Cross-platform JS - ES6 Generators & Promises - promises - chain

- calling `then` on the returned promise creates a new *promise*
- if this promise is now resolved successfully
  - *we can then register an additional callback*
- we may now chain as many `then` methods as necessary
- create a sequence of promises
  - *each resolved &c. one after another*
- instead of creating deeply nested callbacks
  - *simply chain such methods to our initial resolved promise*
- to catch an error we may chain a final `catch` call
- to catch an error for the overall chain
  - *use the `catch` method for the overall chain*

```
getJSON().then()
.then()
.then()
.catch((err) => {
  // Handle any error that occurred in any of the previous promises in the chain.
    console.log('error found = ', err); // not much to show due to return of jsonp from fl
});
```

- if a failure occurs in any of the previous promises
  - *the `catch` method will be called*

Cross-platform JS - ES6 Generators & Promises - promises - wait for multiple promises

- promises also make it easy to wait for multiple, independent asynchronous tasks
- with `Promise.all`, we may wait for a number of promises

```
// wait for a number of promises - all
Promise.all([
// call getJSON with required URL, `then` method for resolve object, and `catch` for error
getJSON("notes.json"),
getJSON("metadata.json")]).then(response => {
  // check return value from promise...response[0] = notes.json, response[1] = metadata.js
  if (response[0] !== null) {
        console.log("response obtained");
    console.log("notes = ", JSON.stringify(response[0]));
    console.log("metadata = ", JSON.stringify(response[1]));
    }
}).catch((err) => {
  // Handle any error that occurred in any of the previous promises in the chain.
    console.log('error found = ', err); // not much to show due to return of jsonp from fl
});
```

- order of execution for tasks doesn't matter for `Promise.all`
- by using the `Promise.all` method
  - *we are simply stating that we want to wait...*
- `Promise.all` accepts an array of promises
  - *then creates a new promise*
  - *promise will resolve successfully when all passed promises resolve*
- it will reject if a single one of the passed promises fails
- return promise is an array of succeed values as responses
  - *i.e. one succeed value for each passed in promise*

Cross-platform JS - ES6 Generators & Promises - promises - racing promises

- we may also setup competing promises
  - *with an effective prize to the first promise to resolve or reject*
  - *might be useful for querying multiple APIs, databases, &c.*

```
Promise.race(
  [
  // call getJSON with required URL, `then` method for resolve object, and `catch` for err
  getJSON("notes.json"),
  getJSON("metadata.json")]).then(response => {
    if (response !== null) {
        console.log(`response obtained - ${response} won...`);
    }
  }).catch((err) => {
    // Handle any error that occurred in any of the previous promises in the chain.
    console.log('error found = ', err); // not much to show due to return of jsonp from fl
  });
);
```

- method accepts an array of promises
  - *returns a completely new resolved or rejected promise*
  - *returns for the first resolved or rejected promise*

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API

- MDN - Fetch API

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API - Example

basic usage

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

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

- Demo - Fetch API - basic usage

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API - Example

catching errors

```
/*
* fetch-basic-error1.js
* basic example usage of Fetch API...chain `catch` to `then` for error handling
*/

fetch('./assets/item.json')
  .then(response => {
    // reactions passed to `then` used to handle fulfillment of a promise
    return response.json();
  })
  .then(myJSON => {
    console.log(myJSON);
  })
  .catch(err => {
    // reactions passed to `catch` executed with a rejection reason...
    console.log(`error detected - ${err}`);
  });
```

- Demo - Fetch API - catching errors

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API - Example

Fetch with Promise all

```
/*
* fetch-promise-all.js
* basic example usage of Promise.all...using Fetch API
*/

Promise
  .all([
    fetch('./assets/items.json'),
    fetch('./assets/notes.json')
  ])
  .then(responses =>
    Promise.all(responses.map(res => res.json()))
  ).then (json => {
    console.log(json);
  });
```

- Demo - Fetch API - Promise all

Cross-platform JS - ES6 Generators & Promises - promises - Fetch API - Example

Fetch with Promise race

```
/*
* fetch-promise-race.js
* basic example usage of Promise.race...using Fetch API
*/

Promise
  .race([
    fetch('./assets/items.json'),
    fetch('./assets/notes.json')
  ])
  .then(responses => {
    return responses.json()
  })
  .then(res => console.log(res));
```

- Demo - Fetch API - Promise race

Cross-platform JS - ES6 Generators & Promises - generators

- a *generator* function generates a sequence of values
  - *commonly not all at once but on a request basis*
- generator is explicitly asked for a new value
  - *returns either a value or a response of no more values*
- after producing a requested value
  - *a generator will then suspend instead of ending its execution*
  - *generator will then resume when a new value is requested*

Cross-platform JS - ES6 Generators & Promises - generators - example

```
//generator function
function* nameGenerator() {
  yield "emma";
  yield "daisy";
  yield "rosemary";
}
```

- define a generator function by appending an *asterisk* after the keyword
  - *`function* ()`*
- use the `yield` keyword within the body of the generator
  - *to request and retrieve individual values*
- then consume these generated values using a standard loop
  - *or perhaps the new `for-of` loop*

Cross-platform JS - ES6 Generators & Promises - generators - iterator object

- if we make a call to the body of the generator
  - *an iterator object will be created*
- we may now communicate with and control the generator using the iterator object

```
//generator function
function* NameGenerator() {
  yield "emma";
}
// create an iterator object
const nameIterator = NameGenerator();
```

- iterator object, `nameIterator`, exposes various methods including the `next` method

Cross-platform JS - ES6 Generators & Promises - generators - iterator object - next()

- use `next` to control the iterator, and request its next value

```
// get a new value from the generator with the 'next' method
const name1 = nameIterator.next();
```

- `next` method executes the generator's code to the next yield expression
- it then returns an object with the value of the yield expression
  - *and a property `done` set to false if a value is still available*
- `done` boolean will switch to *true* if no value for next requested yield
- `done` is set to *true*
  - *the iterator for the generator has now finished*

Cross-platform JS - ES6 Generators & Promises - generators - iterate over iterator object

- iterate over the iterator object
  - *return each value per available yield expression*
  - *e.g. use the* `for-of` *loop*

```
// iterate over iterator object
for(let iteratorItem of NameGenerator()) {
  if (iteratorItem !== null) {
    console.log("iterator item = "+iteratorItem+index);
  }
}
```

Cross-platform JS - ES6 Generators & Promises - generators - call generator within a generator

- we may also call a generator from within another generator

```
//generator function
function* NameGenerator() {
  yield "emma";
  yield "rose";
  yield "celine";
  yield* UsernameGenerator();
  yield "yvaine";
}

function* UsernameGenerator() {
  yield "frisby67";
  yield "trilby72";
}
```

- we may then use the initial generator, `NameGenerator`, as normal

References

- Google Dev
  - *Async functions*
- MDN
  - *Async function*
  - *Await*
  - *Generator*
  - *Promises*