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

Fall Semester 2017 - Week 5

Dr Nick Hayward

Extra notes - mobile considerations

Extra design notes will start to be added to the course website, GitHub...e.g.

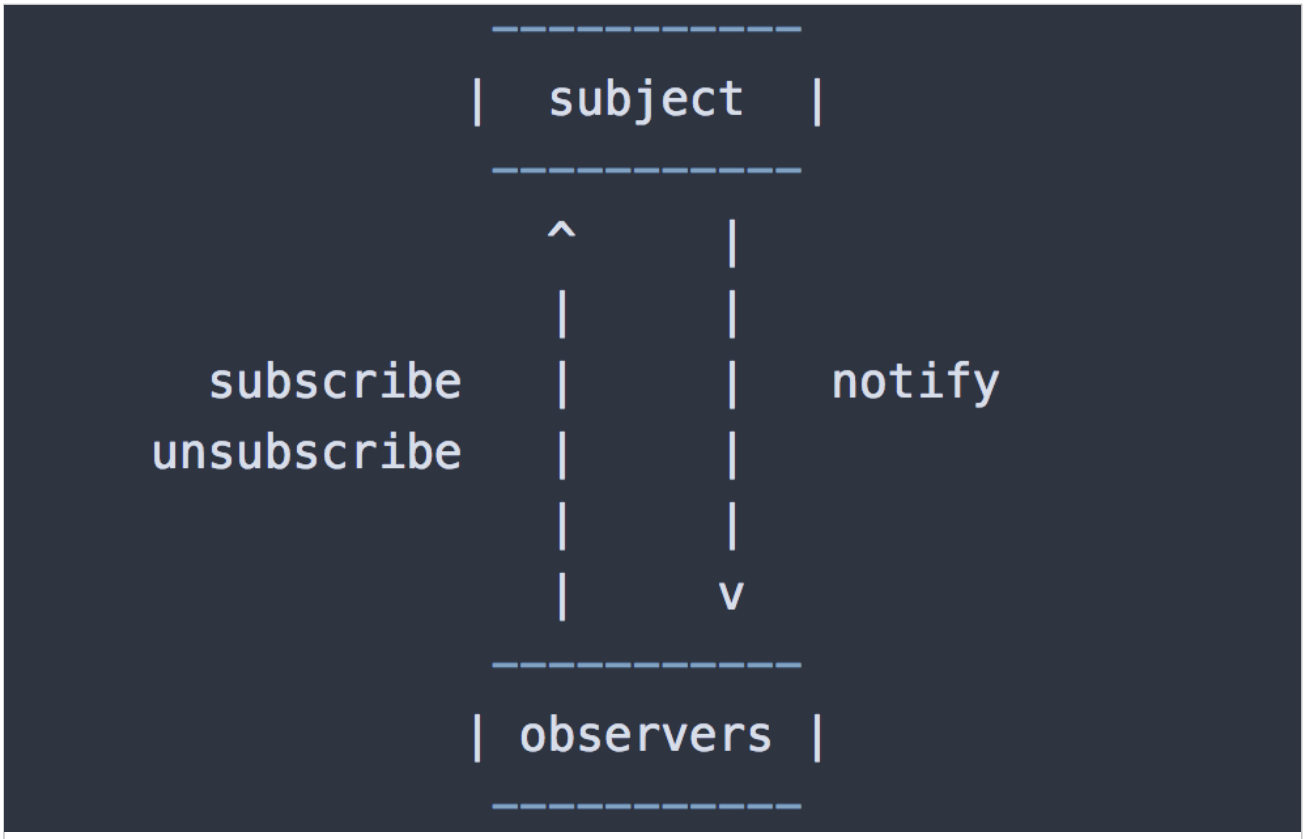
- design mockups
- design and interface
- design and data
- ...

& extra notes on JS &c.

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

```
// 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));
```

Data considerations in mobile apps

- worked our way through Cordova's File plugin
- tested local and remote requests with JSON
- initial considerations for working with LocalStorage
- many other options for data storage in mobile applications
  - *IndexedDB*
  - *hosted NoSQL options, such as Redis and MongoDB*
  - *Firebase*
  - *query hosted remote SQL databases*
  - *and so on...*

Cordova app - IndexedDB

intro

- browser storage wars of recent years
  - *IndexedDB was crowned the winner over WebSQL*
- what do we gain with IndexedDB?
  - *useful option for developers to store relatively large amounts of client-side data*
  - *effectively stores data within the user's webview/browser*
  - *useful storage option for network apps*
  - *a powerful, and particularly useful, indexed based search API*
- IndexedDB differs from other local browser-based storage options
- localStorage is generally well supported
  - *limited in terms of the total amount of storage*
  - *no native search API*
- different solutions for different problems
  - *no universal best fit for storage...*
- browser support for mobile and desktop
  - *Can I use___?*
- Cordova plugin to help with IndexedDB support
  - *MSOpenTech - cordova-plugin-indexeddb*

Cordova app - IndexedDB - data test 2

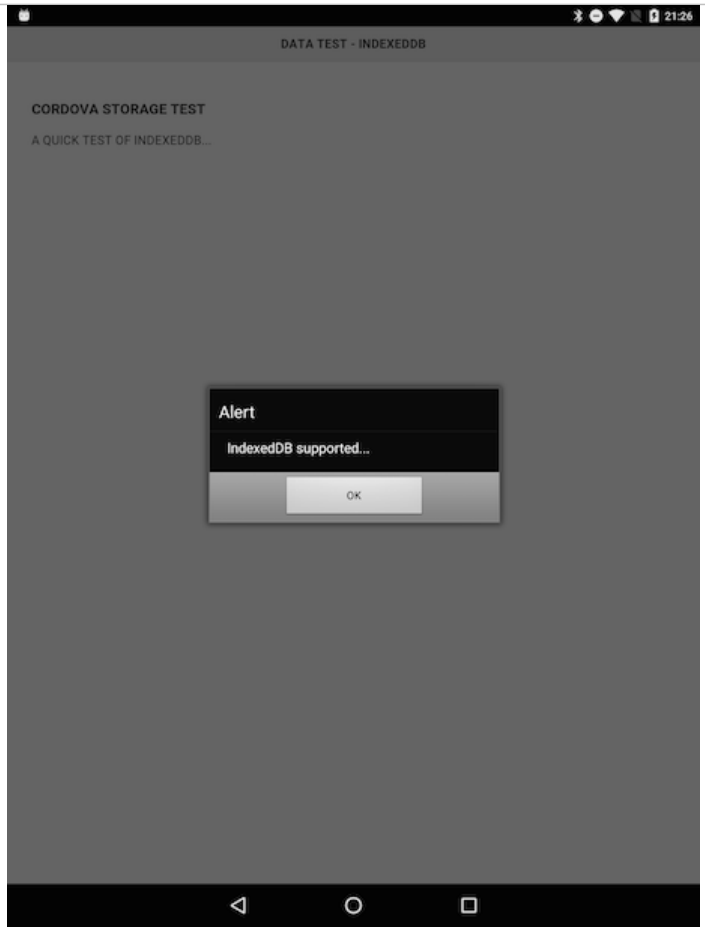
setup and test - part 1

- testing our IndexedDB example with Cordova and Android
- perform our standard test for the `deviceready` event
  - *going to add a check for IndexedDB support and usage*
- in `onDeviceReady()` function
  - *add a quick check for IndexedDB support in the application's webview*

```
if("indexedDB" in window) {
  console.log("IndexedDB supported...");
} else {
  console.log("No support...");
}
```

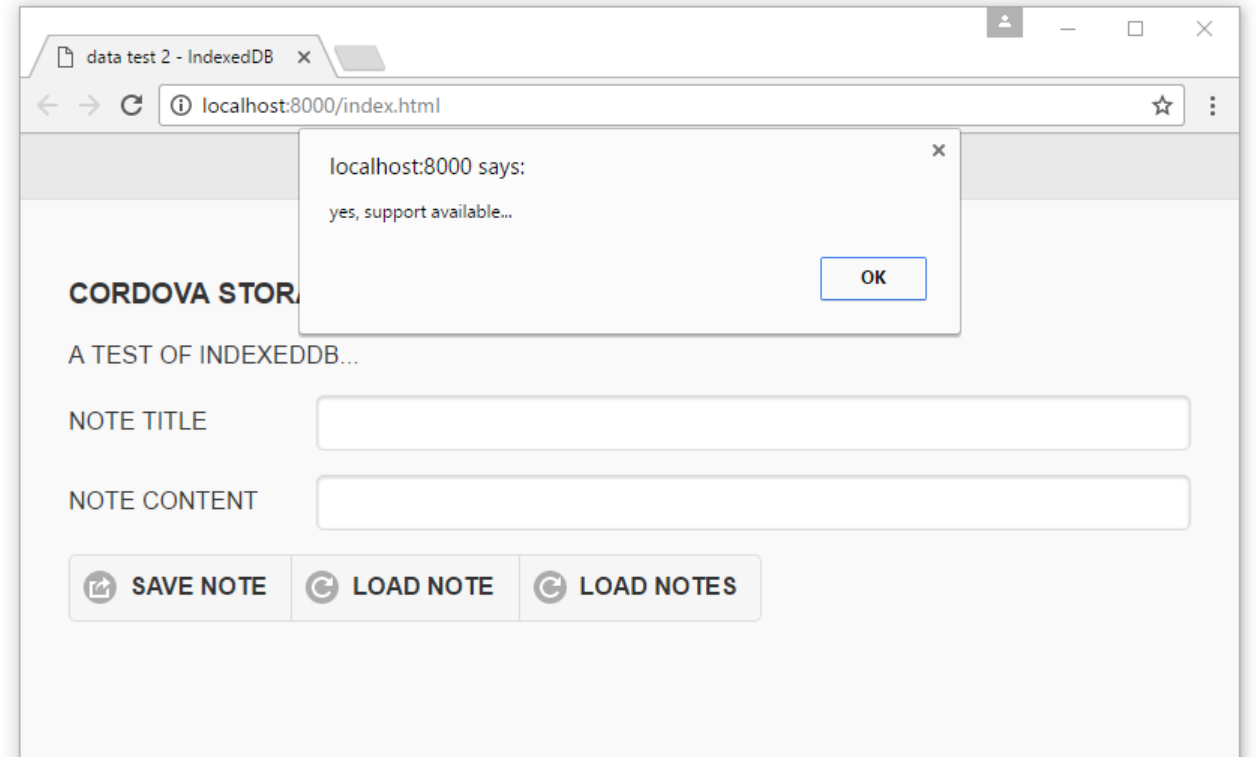
- Android support is available...

Image - IndexedDB Support



DataTest2 - test IndexedDB support in webview

Image - IndexedDB Support



DataTest2 - test IndexedDB support in chrome

Cordova app - IndexedDB - data test 2

setup and test - part 2

- update this check to ensure we have a quick reference later

```
//set variable for IndexedDB support
var indexedDBSupport = false;
//check IndexedDB support
if("indexedDB" in window) {
  indexedDBSupport = true;
  console.log("IndexedDB supported...");
} else {
  console.log("No support...");
}
```

- create initial variable to store the boolean result
- check variable after `deviceready` event has fired and returned successfully

Cordova app - IndexedDB - data test 2

database - part 1 - getting started

- start to build our IndexedDB database
- database is local to the browser,
  - *only available to users of the local, native app*
- IndexedDB databases follow familiar pattern of read and write privileges
  - *eg: browser-based storage options, including localstorage*
- create databases with the same name, and then deploy them to different apps
  - *remain domain specific as well*
- first thing we need to do is create an opening to our database

```
var openDB = indexedDB.open("422test", 1);
```

- creating a variable for our database connection
  - *specifying the name of the DB and a version*
- open request to the DB is an asynchronous operation

Cordova app - IndexedDB - data test 2

database - part 2 - getting started

- open request to the DB is an asynchronous operation
  - *add some useful event listeners to help with our application*
  - *`success`, `error`, `upgradeneeded`, `blocked*
- `upgradeneeded`
  - *event will fire when the DB is first opened within our application*
  - *also if and when we update the version number for the DB*
- `blocked`
  - *fires when a previous or defunct connection to the DB has not been closed*

Cordova app - IndexedDB - data test 2

database - part 3 - create

- test creating a new DB
  - *then checking persistence during application loading and usage*

```
if(indexedDBSupport) {
  var openDB = indexedDB.open("422test",1);
  openDB.onupgradeneeded = function(e) {
      console.log("DB upgrade...");
  }
  openDB.onsuccess = function(e) {
      console.log("DB success...");
      db = e.target.result;
  }
  openDB.onerror = function(e) {
      console.log("DB error...");
      console.dir(e);
  }
}
```

- `console.log()` - outputs a string representation
- `console.dir()` - prints a navigable tree

Image - IndexedDB Support

DataTest2 - test IndexedDB open - first app load

Cordova app - IndexedDB - data test 2

database - part 4 - success

- performed a check to ensure that IndexedDB is supported
  - *if yes, open a connection to the DB*
  - *also added checks for three events, including `upgrade`, `onsuccess`, and `errors`*
- now ready to test the `success` event
  - *event is passed a handler via `target.result`*

```
...
openDB.onsuccess = function(e) {
    console.log("DB success...");
    db = e.target.result;
}
...
```

- handler is being stored in our global variable `db`
- run this test and check log output
  - *outputs initial connection and upgrade status*
  - *then the success output for subsequent loading of the application*

Image - IndexedDB Support

DataTest2 - test IndexedDB open - after first app load

Cordova app - IndexedDB - data test 2

database - part 5 - data stores

- now start building our data stores in IndexedDB
- IndexedDB has a general concept for storing data
  - *known as **Object Stores***
  - *conceptually at least, known as (very) loose database tables*
- within our object stores
  - *add some data, plus a **keypath**, and an optional set of indices (indexes)*
- a **keypath** is a unique identifier for the data
- Indices help us index and retrieve the data
- object stores created during `upgradeneeded` event for the current version
  - *created when the app first loads*
  - *create object stores as part of this `upgradeneeded` event*
- if we want to upgrade our object stores
  - *update version*
  - *upgrade the object store using the `upgradeneeded` event*

Cordova app - IndexedDB - data test 2

database - part 6 - data stores

- update our upgrade event to include the creation of our required object stores

```
...
openDB.onupgradeneeded = function(e) {
    console.log("DB upgrade...");
    //local var for db upgrade
    var upgradeDB = e.target.result;
    if (!upgradeDB.objectStoreNames.contains("422os")) {
      upgradeDB.createObjectStore("422os");
      console.log("new object store created...");
    }
}
...
```

- check a list of existing object stores
  - *list of existing object stores available in the property* `objectStoreNames`
- check this property for our required object store using the `contains` method
- if required object store unavailable we can create our new object store
  - *listen for result from this synchronous method*
- as a user opens our app for the first time
  - *the* `upgradeneeded` *event is run*
  - *code checks for an existing object store*
  - *if unavailable, create a new one*
  - *then run the* `success` *handler*

Image - IndexedDB Support

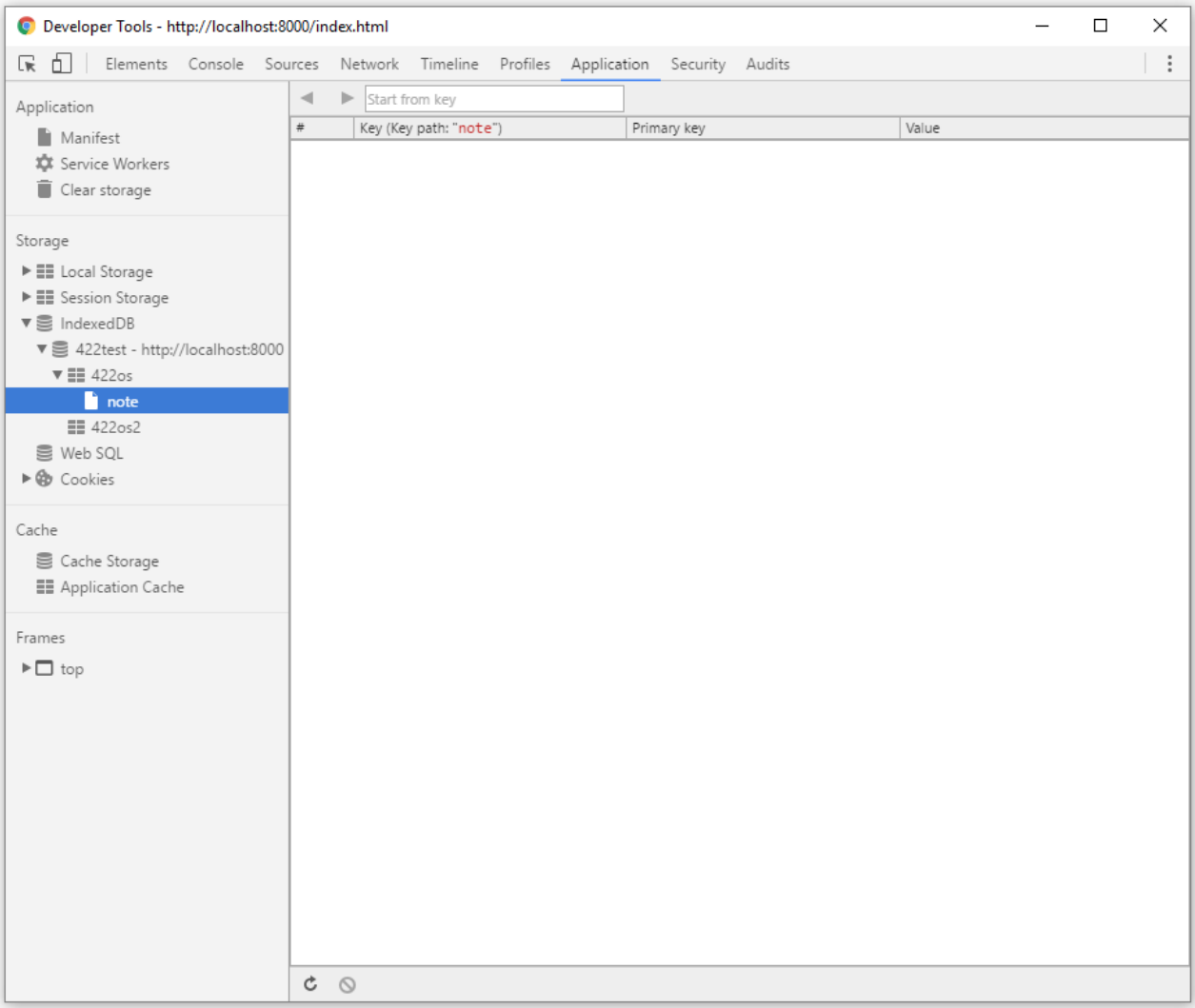
Cordova app - IndexedDB - data test 2

database - part 7 - extra data stores

- start to add further object stores
- can't simply create a new object store due to the `upgradeneeded` event
- increment the version number for the current database
  - *thereby invoking the `upgradeneeded` event*
- reate our new object store using the same pattern

```
var openDB = indexedDB.open("422test",2);
openDB.onupgradeneeded = function(e) {
  console.log("DB upgrade...");
  //local var for db upgrade
  var upgradeDB = e.target.result;
  if (!upgradeDB.objectStoreNames.contains("422os")) {
    upgradeDB.createObjectStore("422os");
    console.log("new object store created...");
  }
  if (!upgradeDB.objectStoreNames.contains("422os2")) {
    upgradeDB.createObjectStore("422os2");
    console.log("new object store 2 created...");
  }
}
```

Image - IndexedDB Support



DataTest2 - test IndexedDB - initial object stores

Cordova app - IndexedDB - data test 2

database - part 8 - add data

- our database currently has two object stores
  - *now start adding some data for our application*
- IndexedDB allows us to simply store our objects in their default structure
  - *simply store JavaScript objects directly in our IndexedDB database*
- use transactions when working with data and IndexedDB
- transactions help us create a bridge between our app and the current database
  - *allowing us to add our data to the specified object store*
- a transaction includes two arguments
  - *first for the object store*
  - *second is the type of transaction*
  - *choose either* `readonly` *or* `readwrite`

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

Cordova app - IndexedDB - data test 2

database - part 9 - add data

- use transaction to retrieve object store for our data
  - *requesting the `422os` in this example*

```
var dataStore = dbTransaction.objectStore("422os");
```

- add some data using the new `datastore`

```
// note
var note = {
  title:title,
  note:note
}
// add note
var addRequest = dataStore.add(note,key);
```

- for each object we can define the underlying naming schema
  - *best fit our applications*
- then add our object, with an associated key, to our `dataStore`

Cordova app - IndexedDB - data test 2

database - part 10 - add data

- now added an object to our object store
- request is asynchronous
  - *attach additional handlers for returned result*
  - *add a success and error handler*

```
// success handler
addRequest.onsuccess = function(e) {
  console.log("data stored...");
  // do something...
}
// error handler
addRequest.onerror = function(e) {
console.log(e.target.error.name);
// handle error...
}
```

Cordova app - IndexedDB - data test 2

database - part 11 - add data

- add a form for the note content and title
- set a save button to add the note date to the IndexedDB

```
<form id="noteForm">
  <div class="ui-field-contain">
    <label for="noteName">Note Title</label>
    <input type="text" id="noteName" name="noteName"></input>
  </div>
  <div class="ui-field-contain">
    <label for="noteContent">Note Content</label>
    <input type="text" id="noteContent" name="noteContent"></input>
  </div>
  <div data-role="controlgroup" data-type="horizontal">
    <input type="button" id="saveNote" data-icon="action" value="Save Note" data-inline="true"/>
  </div>
</form>
```

- bind event handler to save button for click
  - *submit add request to IndexedDB*
  - *store object data*

Cordova app - IndexedDB - data test 2

database - part 12 - add data handlers

- now add our event handler for the save button
- handler gets note input from note form
- passes the data to the `saveNote()` function

```
// handler for save button
$("#saveNote").on("tap", function(e) {
  e.preventDefault();
  var noteTitle = $("#noteName").val();
  var noteContent = $("#noteContent").val();
  saveNote(noteTitle, noteContent);
});
```

Cordova app - IndexedDB - data test 2

database - part 13 - add data handlers

```
//save note data to indexeddb
function saveNote(title, content){
  //define a note
  var note = {
    title:title,
    note:content
  }
  // create transaction
  var dbTransaction = db.transaction(["422os"],"readwrite");
  // define data object store
  var dataStore = dbTransaction.objectStore("422os");
  // add data to store
  var addRequest = dataStore.add(note,1);
  // success handler
  addRequest.onsuccess = function(e) {
    console.log("data stored...");
    // do something...
  }
  // error handler
  addRequest.onerror = function(e) {
  console.log(e.target.error.name);
  // handle error...
  }
}
```

Image - IndexedDB Support

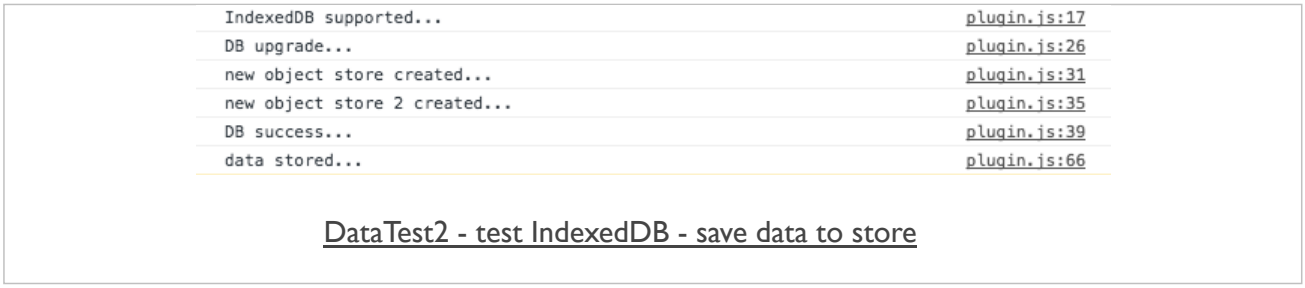
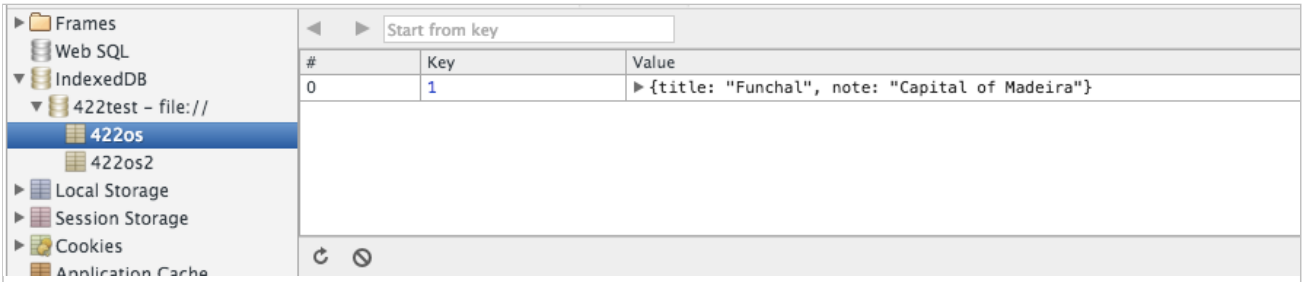
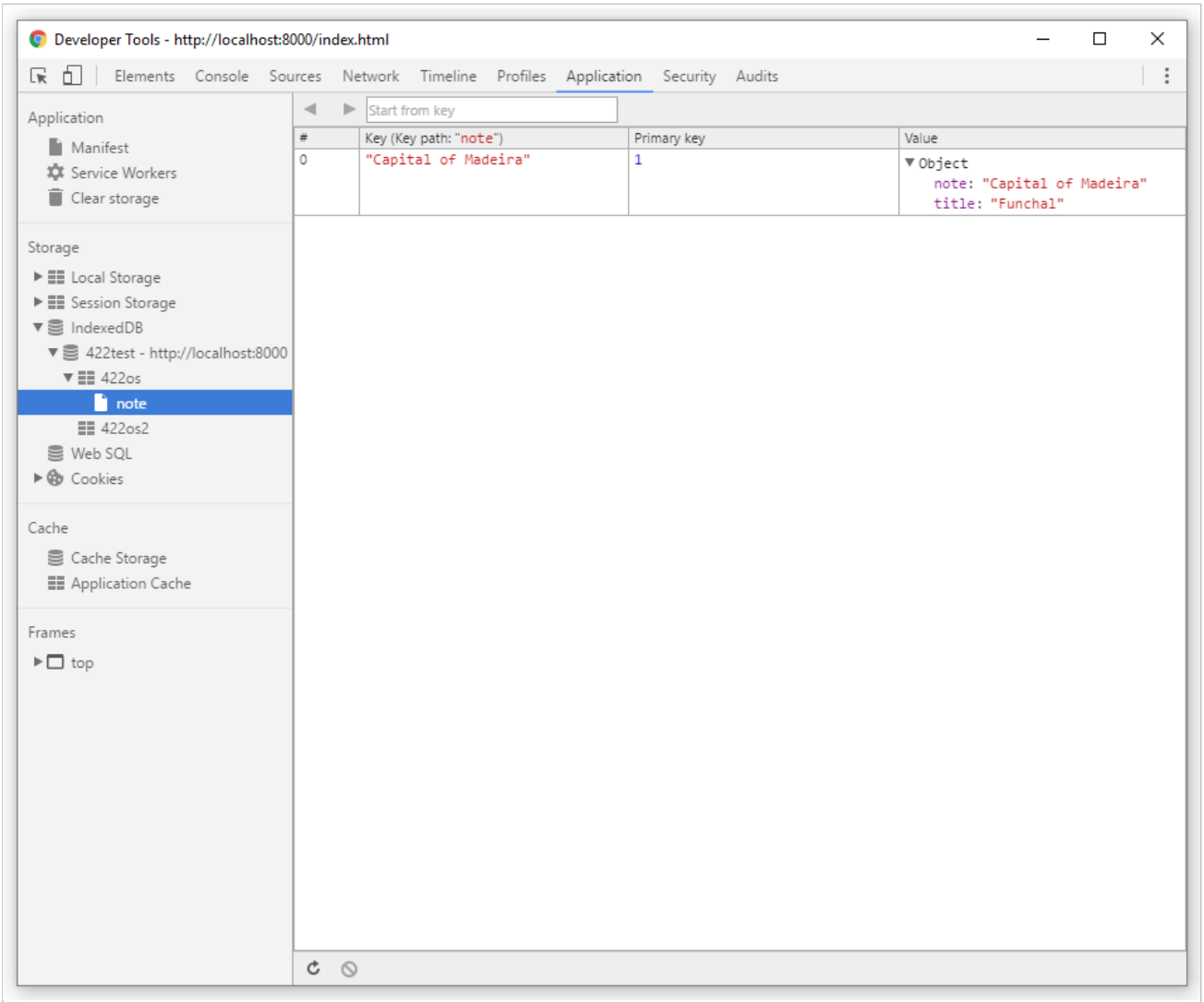


Image - IndexedDB Support



DataTest2 - test IndexedDB - save data to store 2

Image - IndexedDB Support



DataTest2 - test IndexedDB - save data to store 3

Cordova app - IndexedDB - data test 2

database - part 14 - multiple notes

- now created our IndexedDB
- created the object store
- setup the app's HTML and form
- and saved some data to the database...
- update our application to allow a user to add multiple notes to the database
- currently setting our key for a note in the `saveNote()` function
  - *add another note, we get a constraint error output to the console*
  - *we're trying to add a note to an existing key in the database*
- need to update our logic for the app
  - *to allow us to work more effectively with* ***keys***

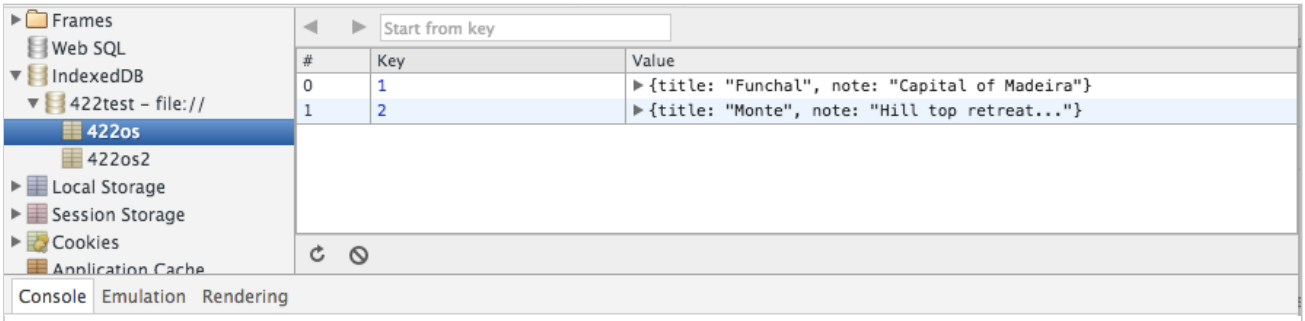
Cordova app - IndexedDB - data test 2

database - part 15 - keys

- keys in IndexedDB often considered similar to primary keys in SQL...
  - *a unique reference for our data objects*
- traditional databases can include tables without such keys
  - ***NB:** every object store in IndexedDB needs to have a **key***
  - *able to use different types of keys for such stores*
- first option for a key is simply to create and add a key ourselves
  - *could programatically create and update these keys*
  - *helps maintain unique ID for keys*
- could also provide a **keypath** for such keys
  - *often based on a given property of the passed data...*
  - *still need to ensure our key is unique*
- other option is to use a key generator within our code
  - *similar concept to SQL* `auto-increment`

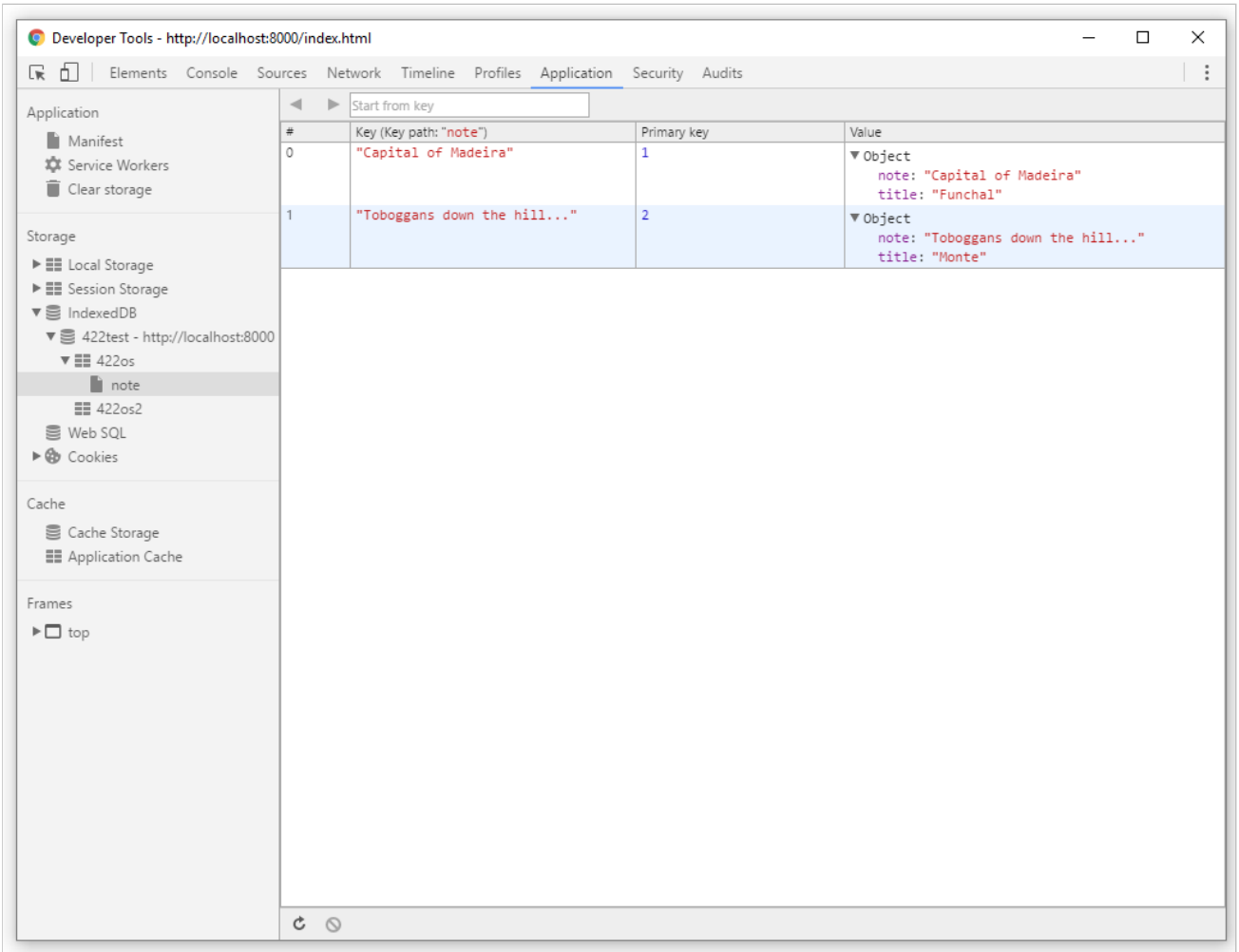
```
db.createObjectStore("422os", { autoIncrement: true });
```

Image - IndexedDB Support



DataTest2 - test IndexedDB - unique keys

Image - IndexedDB Support



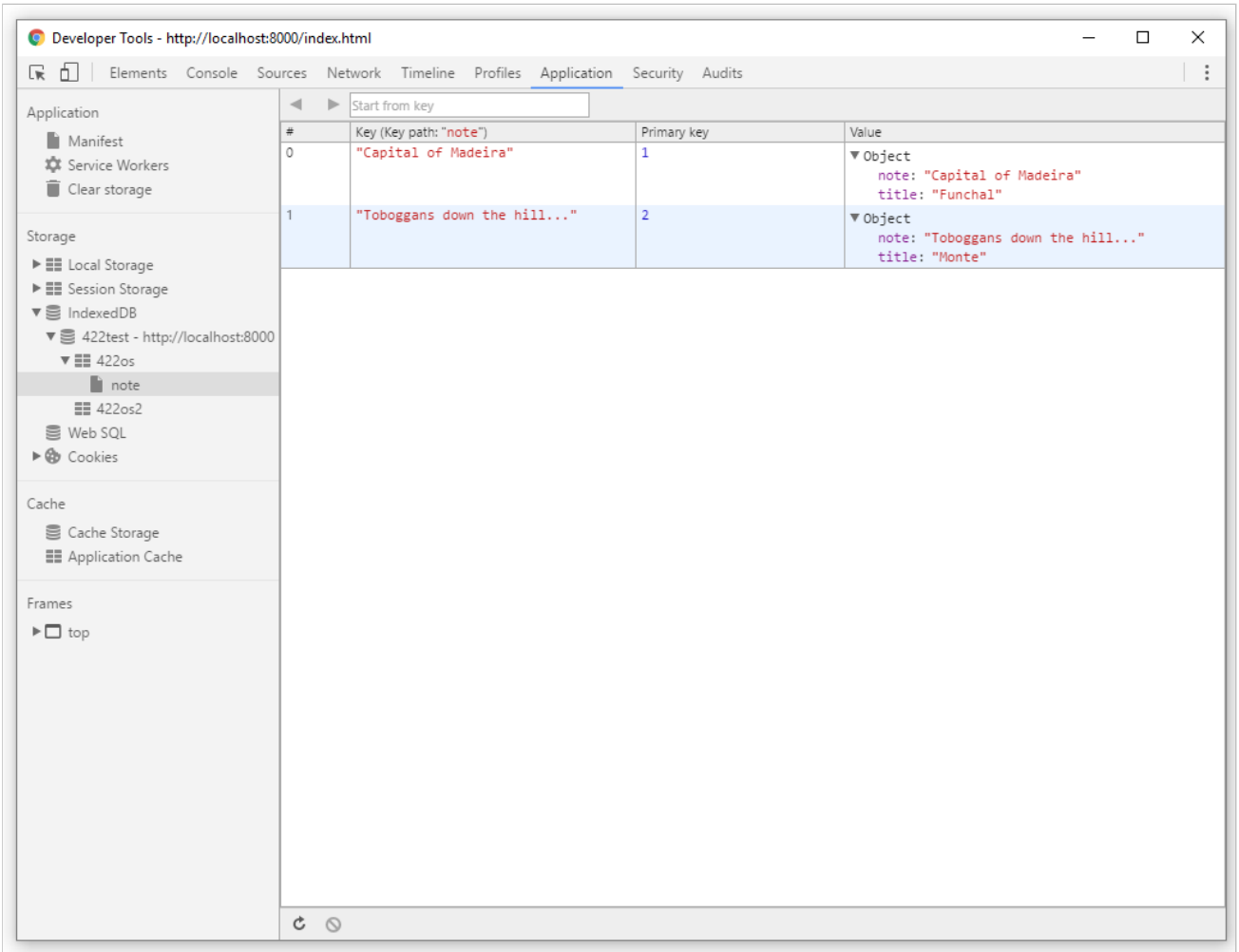
DataTest2 - test IndexedDB - unique keys 2

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



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

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 402cursor = 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*

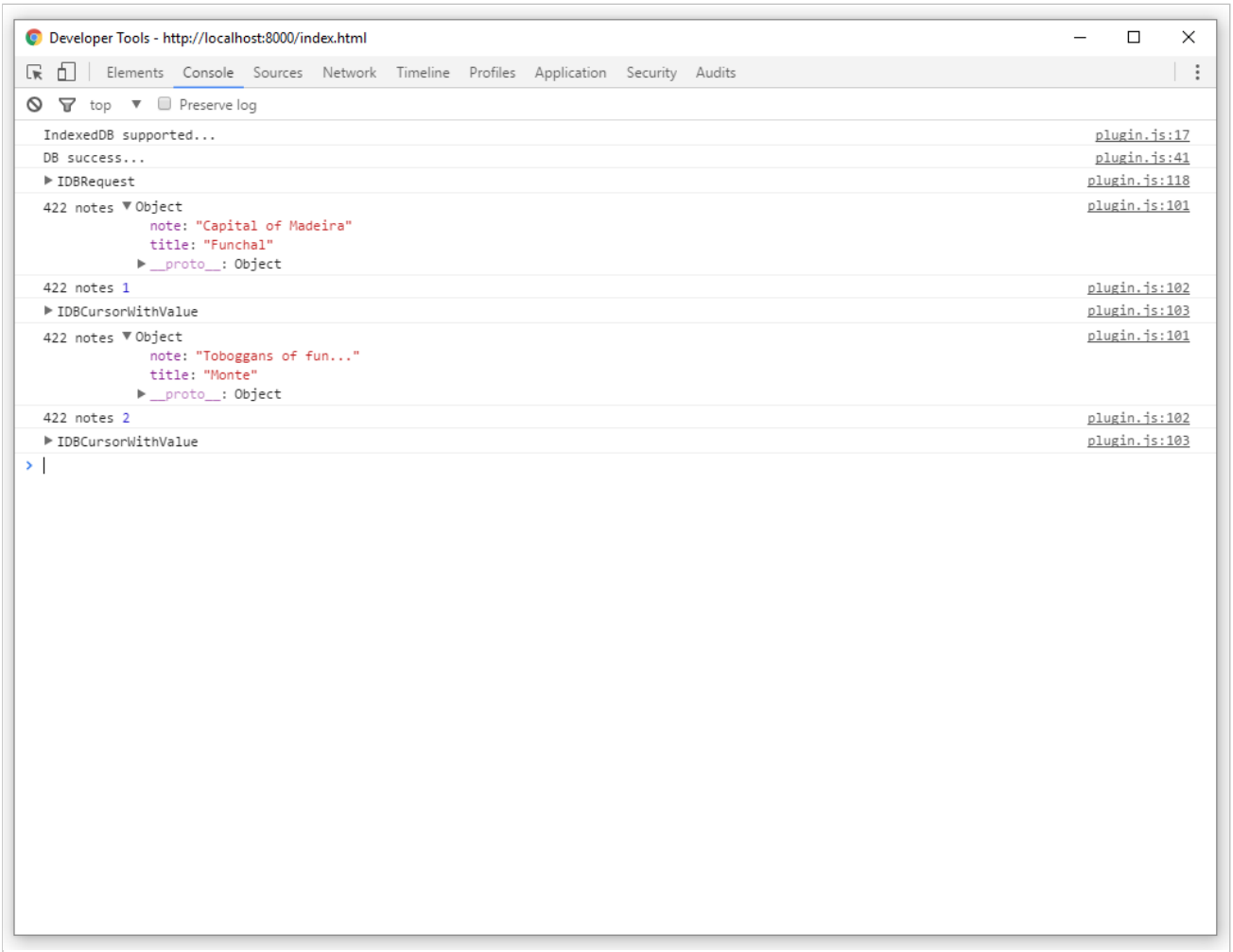
Cordova app - IndexedDB - data test 2

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



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

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");
```

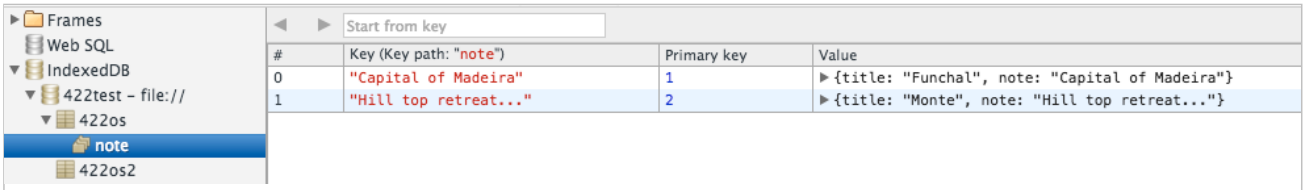
- we can then request some values using a standard `get` method with this index

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

Image - IndexedDB Support

DataTest2 - test IndexedDB - query index

Image - IndexedDB Support



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*

References

- Cordova API
  - *Storage*
- GitHub
  - *cordova-plugin-indexeddb*
  - *cordova-plugin-websql*
- MDN
  - *IndexedDB*
  - *Web APIs - FileError*
- W3
  - *Web storage specification*