University of Sheffield

COM3/6504 The Intelligent Web

Lecture 4: Web Browser Persistence

Andrew Stratton
a.stratton@sheffield.ac.uk

Persistence options

Options for web persistence can mainly be split into:

1. Application Server based. E.g.
   - NodeJS, Java, Golang, C#, Ruby, PHP
2. Database Service (server) - directly accessed through JavaScript, e.g.
   - CouchDB, Firebase

     N.B. There can be SERIOUS security concerns with this approach…
3. Client Browser based storage. Typically attached to a domain. e.g.
   - **Cookies**
   - ~~Parameters~~ - can be bookmarked
   - Cache (we'll come back to this…) - v useful for offline
   - **Web Storage**
   - ~~WebSQL~~ Deprecated
   - **IndexedDB**

Browser Storage (persistence)

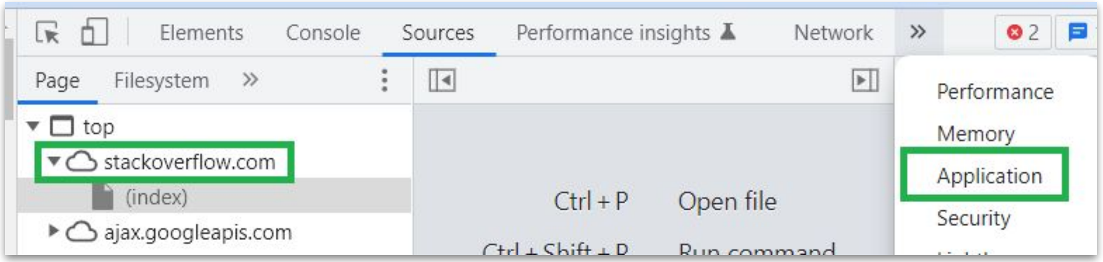
By default **very insecure**

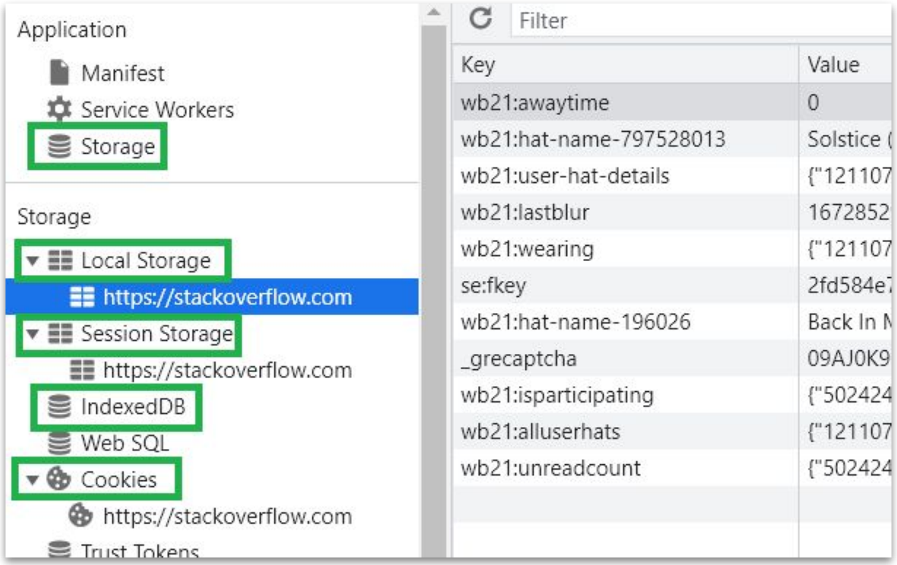
- e.g. viewing stackoverflow in Chrome -> Developer Tools
  - Or `Ctrl+Shift+I`

Check out Application >> Storage

- Cookies
- Local|Session Storage
- IndexedDB

These are all just strings … and easily accessed





Key	Value
wb21:awaytime	0
wb21:hat-name-797528013	Solstice (
wb21:user-hat-details	{"121107
wb21:lastblur	1672852
wb21:wearing	{"121107
se:fkey	2fd584e
wb21:hat-name-196026	Back In N
_grecaptcha	09AJ0K9
wb21:isparticipating	{"502424
wb21:alluserhats	{"121107
wb21:unreadcount	{"502424

University of Sheffield

Cookies

Key features

- Associated with a domain
  - e.g. sheffield.ac.uk
- Also called HTTP/web/browser/internet cookies
- Since ~1996 for most browsers
- Stored on the client device (browser)
  - Are sent to/from server …
- Stored under a single key as text/string - i.e. Key Value pair
  - No direct object storage - can use JSON to serialize/deserialize arrays, objects, etc.
- Common uses include authentication :), shopping carts :), tracking :|

Split into two types:

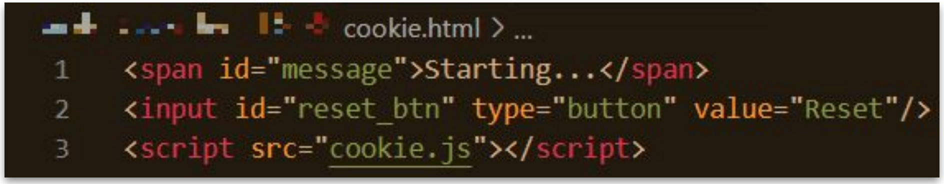
- Session - don’t have an expiration date - these are the ‘default’
- Persistent - have a specified ‘life’

Session Cookie example - visit counter

Creating 'visit counter' to show how often the page has been viewed before

- In a specific browser ….

Start with (incomplete) Html to include the JavaScript:



cookie.html > ...

```html
<span id="message">Starting...</span>
<input id="reset_btn" type="button" value="Reset"/>
<script src="cookie.js"></script>
```

Cookie values should be encoded/decoded - not shown here for brevity

Note: You will (very likely) need to serve this through an application server - e.g. NodeJS

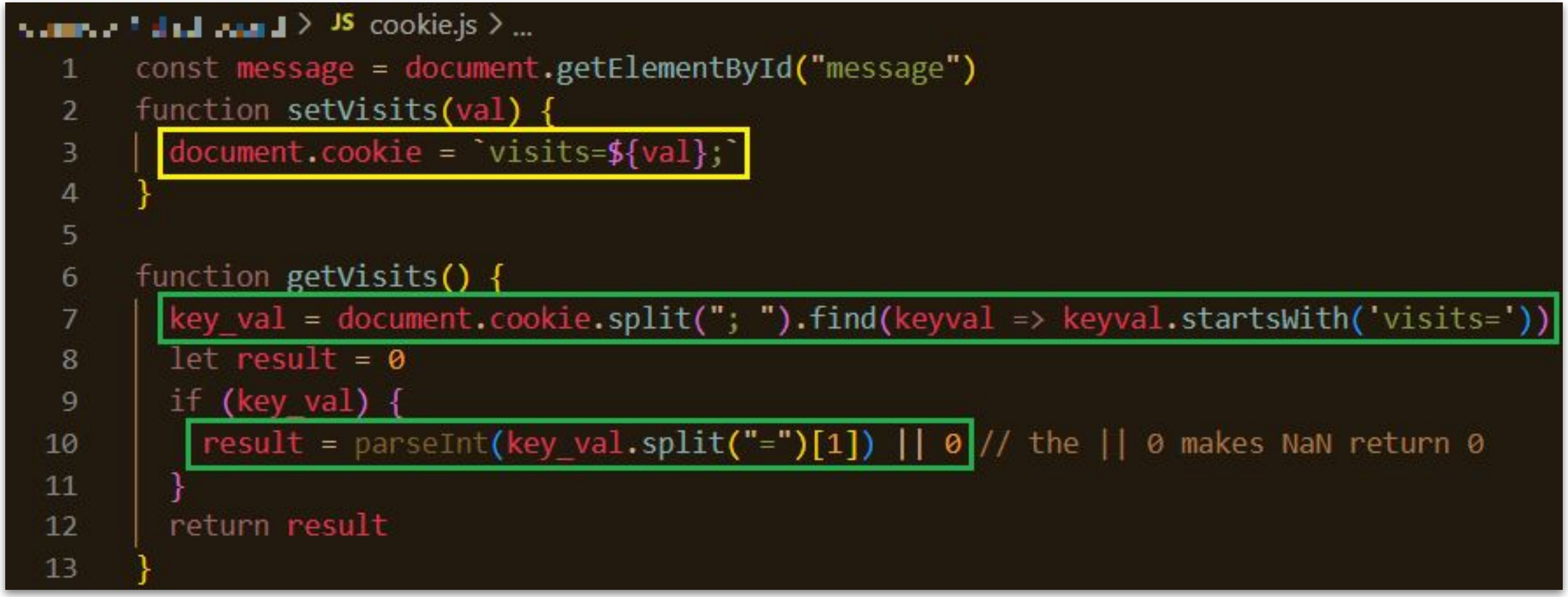
- web hosting will also work …

Simple get/set

The standard cookie get/set/delete is really poor :(

Below sets the cookie key with the (string) value - without encoding

N.B. Get only works assuming that "; " is not in the value (or key) :(



cookie.js > ...

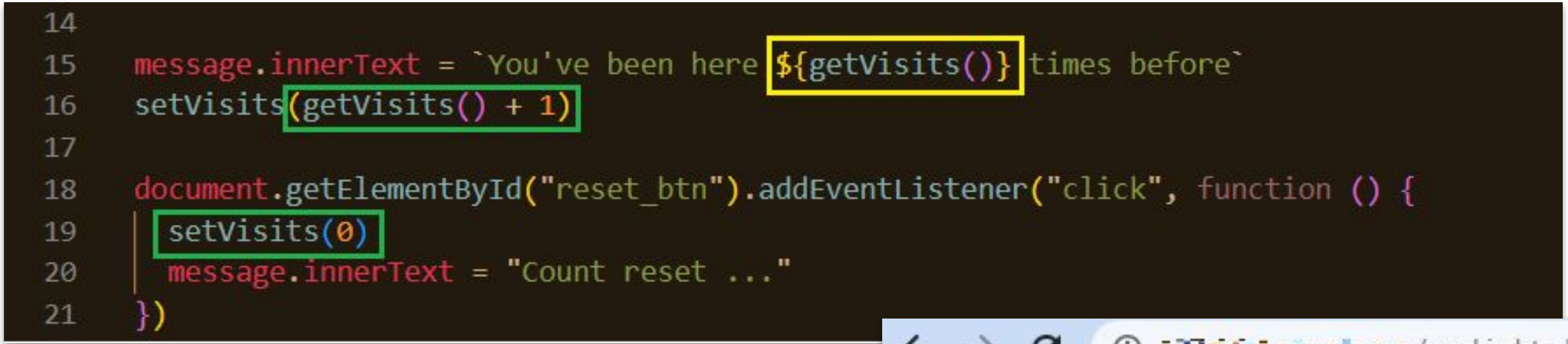
```javascript
const message = document.getElementById("message")
function setVisits(val) {
  document.cookie = `visits=${val};`
}

function getVisits() {
  key_val = document.cookie.split("; ").find(keyval => keyval.startsWith('visits='))
  let result = 0
  if (key_val) {
    result = parseInt(key_val.split("=")[1]) || 0 // the || 0 makes NaN return 0
  }
  return result
}
```

N.B: the `backticks` are not quotes - they are (ES6) template strings

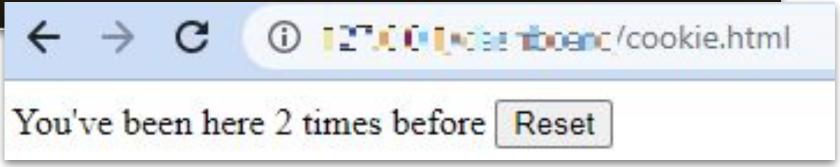
Add output and reset

Note how click listener is added with an anonymous callback function

- You will be seeing a lot of these …



```
message.innerText = `You've been here ${getVisits()} times before`
setVisits(getVisits() + 1)

document.getElementById("reset_btn").addEventListener("click", function () {
    setVisits(0)
    message.innerText = "Count reset ..."
})
```



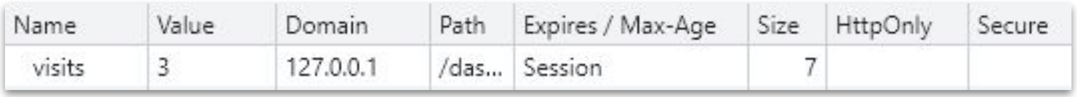
```
[redacted]/cookie.html

You've been here 2 times before [Reset]
```

- Run - then refresh (Ctrl-r) to increment the visits

Developer Tools - invaluable

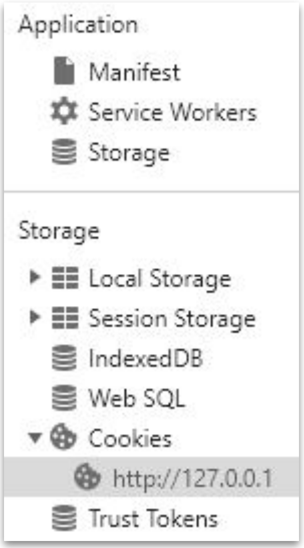
Open (Chrome) developer tools:

1. locate the Application tab
2. Then choose storage >> cookies
   - The example shows localhost development server >>>>>>>>>>>>>>>>>>>>
3. You can see the cookie value:



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure
visits	3	127.0.0.1	/das...	Session	7		

4. You (and anyone else) can also edit it :(



Cookie options

You can add attributes, as `` `key=value;` `` pairs, including:

- Expires/Max-Age - allows the cookie to live longer than a session
- Secure (no value) - can only be accessed through https connections
    - Or localhost/127.0.0.1 for development (I think?)
- Domain/Path - beware this can cause issues
    - Especially avoid (super cookie) domains of '.co.uk' since most browsers will block this
    - N.B. Setting to the default domain/path is NOT the same as letting it default to the same :(
- HttpOnly (no value) - cannot be accessed by JavaScript in browser
    - These cookies must be set in the server application

The first two maybe worth using - other attributes are for specific needs …

N.B. We do not expect (<u>or want</u>) you to use cookies for this module - they are too limited

- ***but you do need to know what they are and how you might use them***

Web Storage

Key features

- 10+ years availability in most browsers
- Stores text/string under a single key - i.e. Key Value pair
  - No direct object storage - can use JSON to serialize/deserialize arrays, objects, etc.
- Associated with a domain
  - e.g. sheffield.ac.uk

Split into two types:

- localStorage
- sessionStorage

University of Sheffield

Web Storage - Local Storage

Benefits/issues include:

- + Easy
- + Fast
- + Simple key value pair model

- - Simple
- - Ltd space
- - Blocking, i.e. single thread access

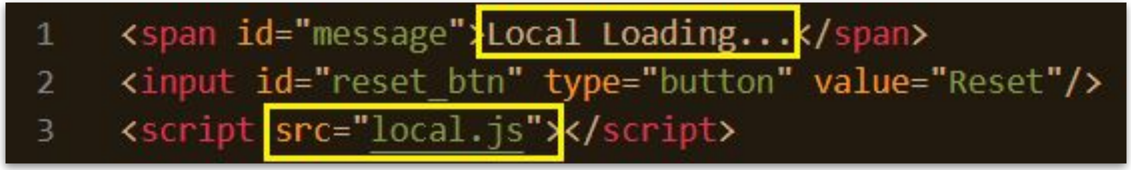
Versus cookies:

Local Storage	**Persistent Cookies**
+ Upto 5MByte	⊝ Upto 4KByte each
⊝ Newer	**+** Legacy better compatibility on **old**er browsers
+ Kept in client (browser)	⊝ Sent to server EVERY request inc, Ajax?!
	⊝ Sent in ‘the clear’ for HTTP (not HTTPS)

Note: Session Storage is very similar, but only for <u>the session</u>

Local Storage example

The Html is hardly changed:



```
<span id="message">Local Loading...</span>
<input id="reset_btn" type="button" value="Reset"/>
<script src="local.js"></script>
```

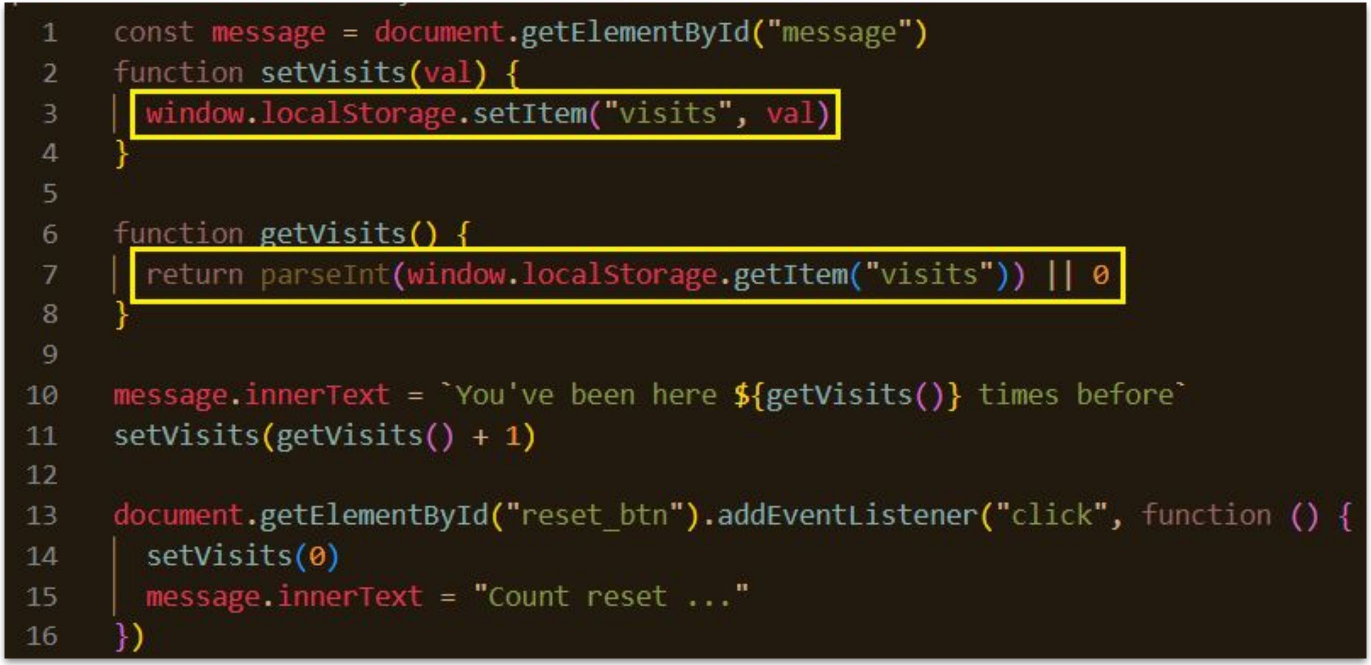
Note: I changed the message so we know if we accidentally just open the cookie version…

BTW - local storage doesn’t need to worry about encoding/decoding string values

Local Storage get/set string

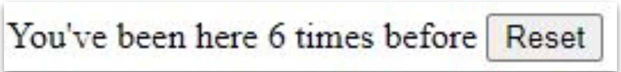
This is **sooo** much simpler:

Note: getItem returns null when not found - which parses as 'NaN' which is false…

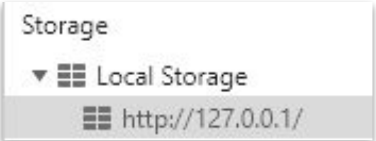


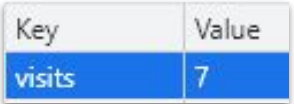
```
const message = document.getElementById("message")
function setVisits(val) {
  window.localStorage.setItem("visits", val)
}

function getVisits() {
  return parseInt(window.localStorage.getItem("visits")) || 0
}

message.innerText = `You've been here ${getVisits()} times before`
setVisits(getVisits() + 1)

document.getElementById("reset_btn").addEventListener("click", function () {
  setVisits(0)
  message.innerText = "Count reset ..."
})
```

Running this …



You've been here 6 times before



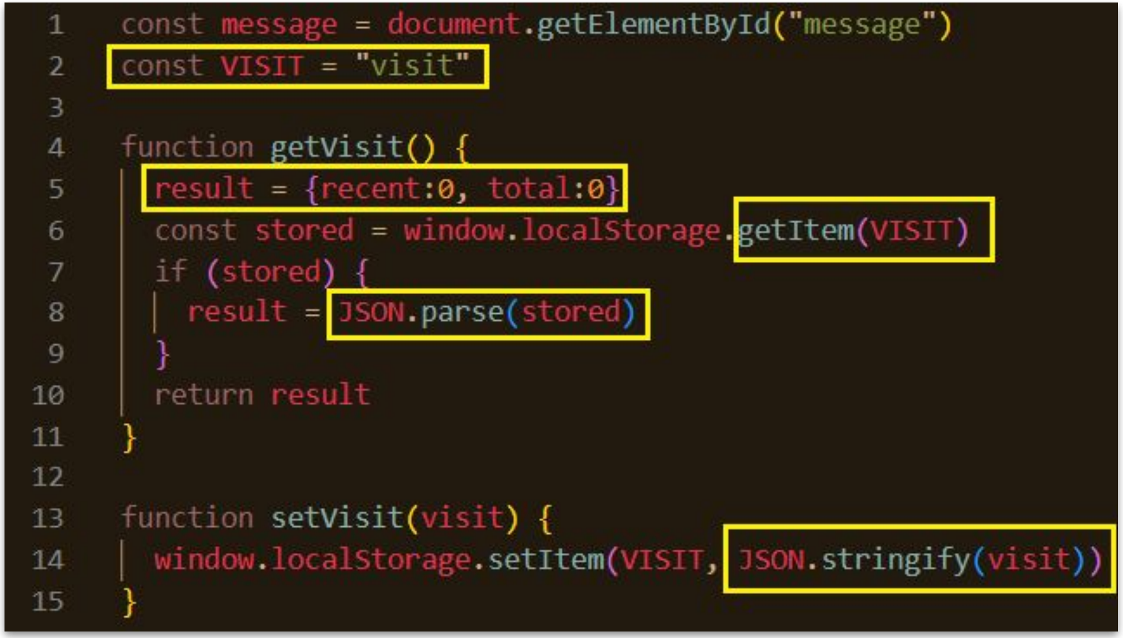


Key	Value
visits	7

Local Object Storage

This would be better if we stored objects :

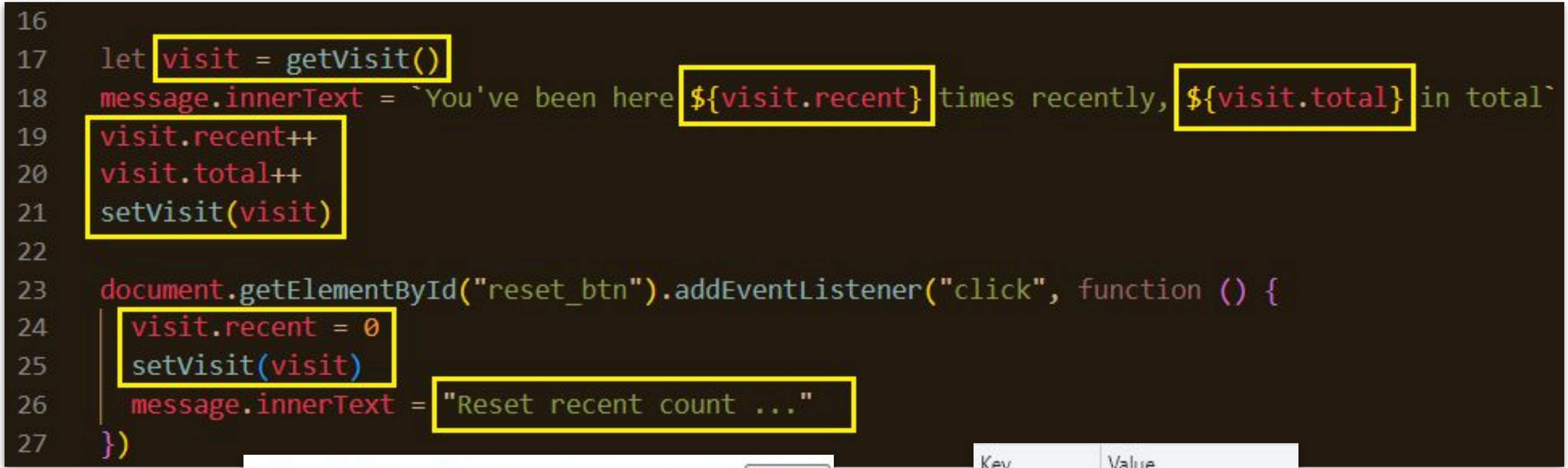
- using a constant now
- the object has two keys (properties)
- must parse into object
- and stringify to string



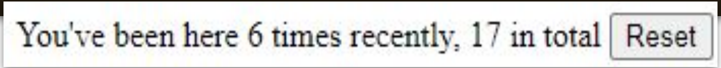
```
const message = document.getElementById("message")
const VISIT = "visit"

function getVisit() {
  result = {recent:0, total:0}
  const stored = window.localStorage.getItem(VISIT)
  if (stored) {
    result = JSON.parse(stored)
  }
  return result
}

function setVisit(visit) {
  window.localStorage.setItem(VISIT, JSON.stringify(visit))
}
```

Object updates

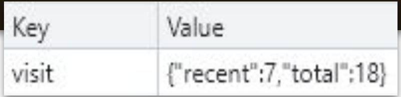
A few changes to handle the two different properties:



```

let visit = getVisit()
message.innerText = `You've been here ${visit.recent} times recently, ${visit.total} in total`
visit.recent++
visit.total++
setVisit(visit)

document.getElementById("reset_btn").addEventListener("click", function () {
  visit.recent = 0
  setVisit(visit)
  message.innerText = "Reset recent count ..."
})
```



You've been here 6 times recently, 17 in total Reset



Key	Value
visit	{"recent":7,"total":18}

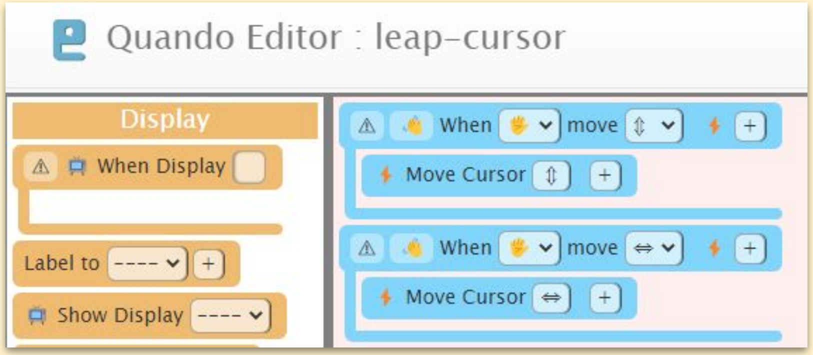
Note how the object is held (still editable)

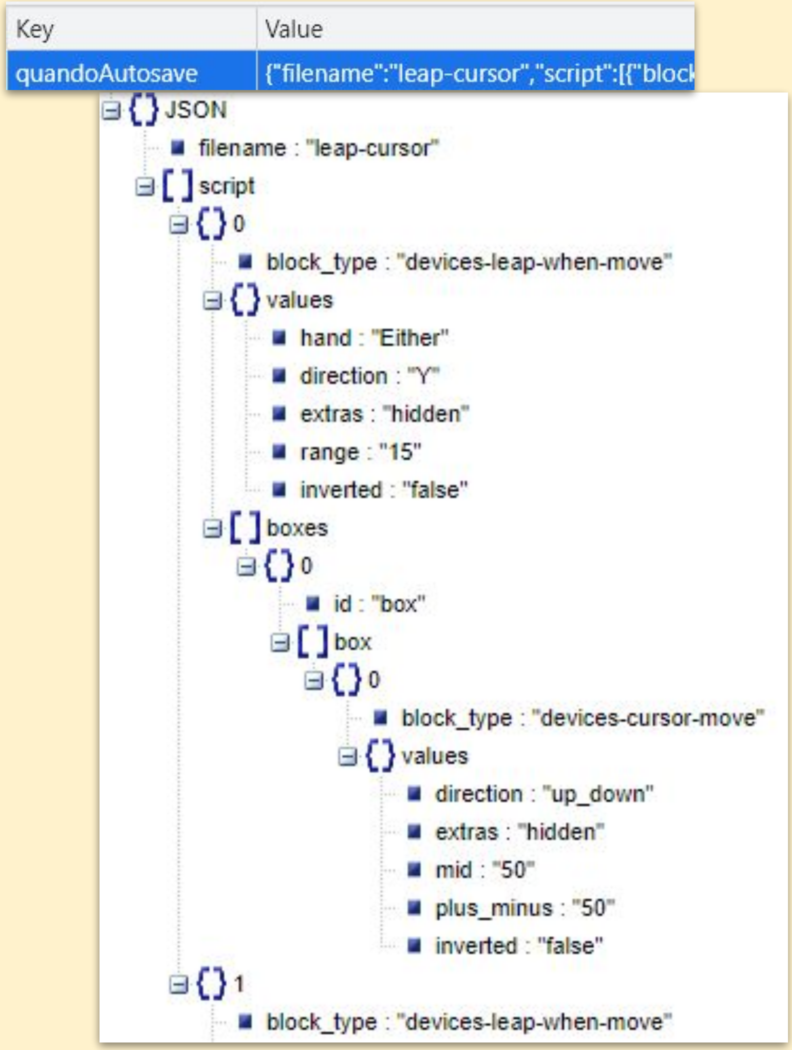
Session Storage is similar - but lifetime is for session …

Example JSON

This is from my ‘Quando’ Visual Programming editor

- the last session is held in localStorage
- the filename for saving is kept
- and the last script created
  - e.g. for shopping this could be a cart …
- this is called from ‘beforeunload’


Quando Editor : leap–cursor



Key	Value
quandoAutosave	{"filename":"leap-cursor","script":[{"block

```
{} JSON
   filename : "leap-cursor"
   [] script
      {} 0
         block_type : "devices-leap-when-move"
         {} values
            hand : "Either"
            direction : "Y"
            extras : "hidden"
            range : "15"
            inverted : "false"
         [] boxes
            {} 0
               id : "box"
               [] box
                  {} 0
                     block_type : "devices-cursor-move"
                     {} values
                        direction : "up_down"
                        extras : "hidden"
                        mid : "50"
                        plus_minus : "50"
                        inverted : "false"
      {} 1
         block_type : "devices-leap-when-move"
```

Short break

Why use addEventListener("click", …) instead of onclick?

onclick, etc., only hold ONE handler

University of Sheffield

IndexedDB

Compared with Web (local) Storage:

IndexedDB	Local Storage
+ Upto 60% of free disk space for domain + persist objects + non blocking - event/promise based + indexed so can search	⊖ Upto 5MByte ⊖ only persist strings ⊖ single thread, blocking ⊖ big array
⊖ More complex API ⊖ Newer with less coverage ⊖ Evolving version/s - v1:2015 +2:2018 v3 in draft	+ Simpler API + Older & more compatible + One solid version

Useful resources:

- https://javascript.info/indexeddb • https://web.dev/indexeddb/
- https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
- https://www.w3.org/TR/IndexedDB/ The definitive specification - really for development of new browsers
- https://caniuse.com/?search=indexeddb - usually worth checking but not very useful for indexedDB

IndexedDB compared with SQL

When connecting to SQL, typically:

1. Connect to database server
   - *(create database)*
2. Select a database
   - *(create table schemas)*
3. On 1+ table/s
4. many of CRUD:
   - Create (insert) record (row) in table/s
   - Read record/s
   - Update record/s
   - Delete record/s (rare)

Note: (once only steps-may be manual)

With IndexedDB

1. *No need - creation is automatic when opened*
2. Open database
   (may create)
3. Select objectStore (like table)
   - *(update if version changed)*
4. many of:
   -

IndexedDB Database/table creation

Remember - unlike SQL:

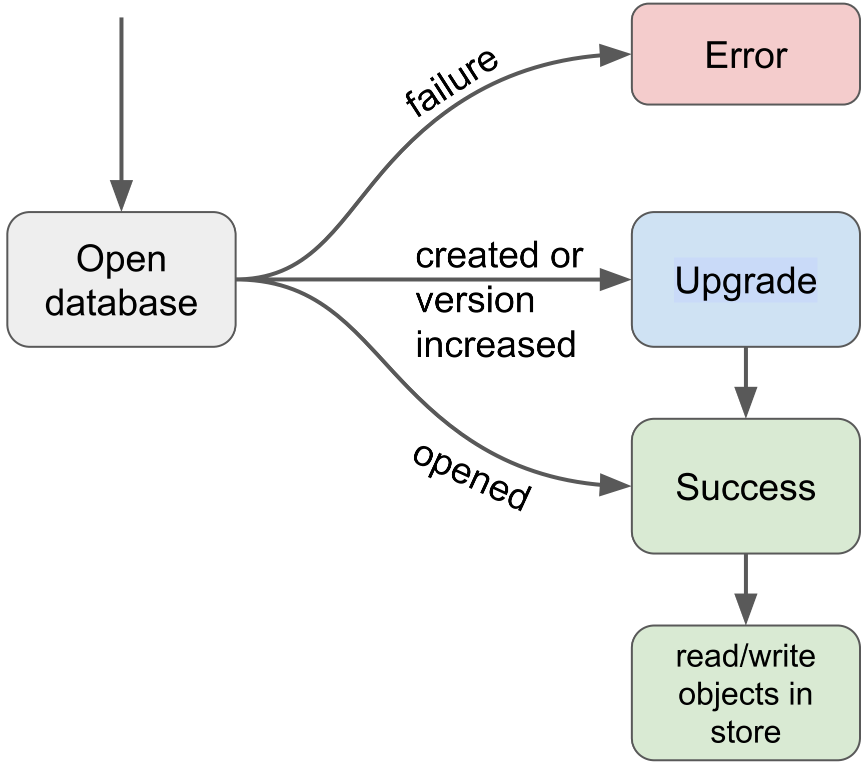
- No need to connect
  - Assigned locally to domain
- No need to create db
  - Open is enough
- No need to create table/s based on schema/s
  - There are no schemas - like NoSql

i.e. all you need to do is open the database :)

Note: I will show 'Vanilla' JavaScript, i.e. not using external libraries - typically ES6

Open states

Remember - Browser JavaScript is single threaded (except for web workers)



E.g. code has older version, not trusted

Can only add/modify Object Stores here
Also called on **creation**/first open
*N.B. version is an **integer**, i.e. 1.9 is 1*

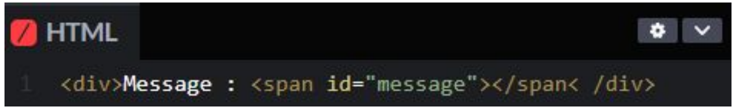
Cannot use transactions before success

MUST read/write objects using transactions

Open DB example

Note: Promises are expected in v3

Html is minimal for clarity:



```html
<div>Message : <span id="message"></span< /div>
```

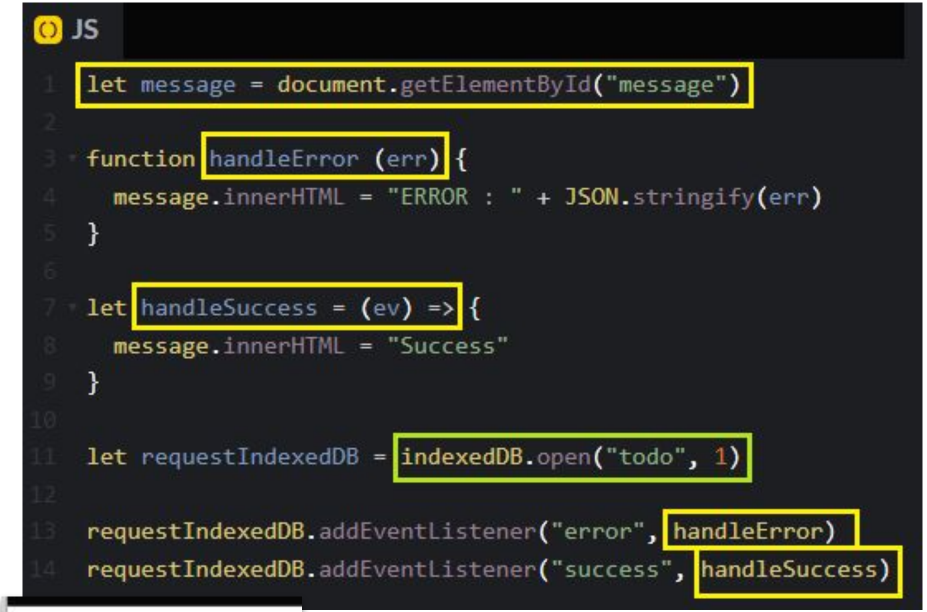
Example JavaScript:

- *indexedDB* is an instance of IDBFactory
- The '1' is the developer version of the DB

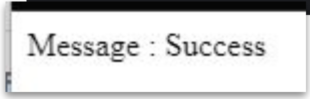
I've used https://codepen.io/pen/ for these examples, if you also do, then

- *Recommend settings >> Behaviour >> **disable** auto update, then use 'Run'*
- *To copy use 'fork' (at bottom)*

Note: don't use codepen for cookies/localStorage



```javascript
let message = document.getElementById("message")

function handleError (err) {
  message.innerHTML = "ERROR : " + JSON.stringify(err)
}

let handleSuccess = (ev) => {
  message.innerHTML = "Success"
}

let requestIndexedDB = indexedDB.open("todo", 1)

requestIndexedDB.addEventListener("error", handleError)
requestIndexedDB.addEventListener("success", handleSuccess)
```



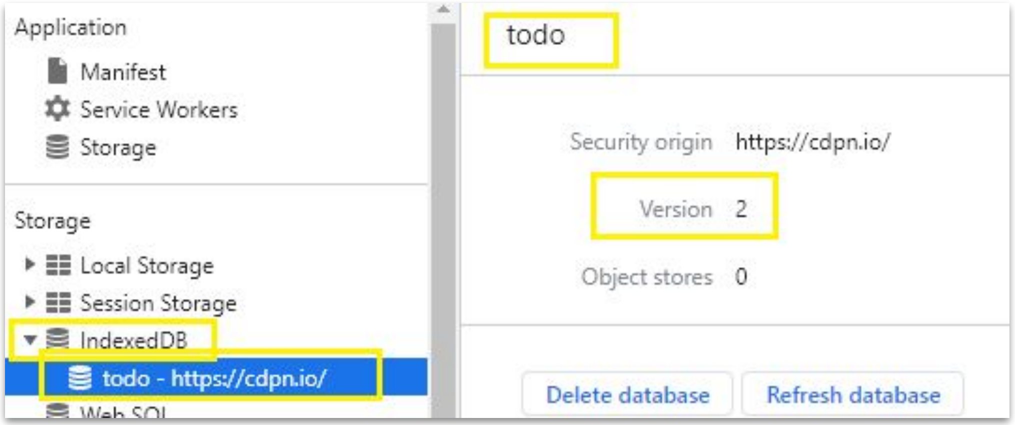
```
Message : Success
```

University of Sheffield

Handling error on open

We can force an error by:

1. Run open DB with version 2 - ‘Success’ (may need to delete db)
2. Use developer tools to see db has been created:



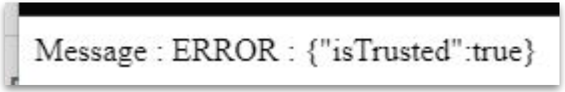
todo

Security origin https://cdpn.io/

Version 2

Object stores 0

3. Then run with version 1. i.e. trying to access **an older version** of developer db



Message : ERROR : {"isTrusted":true}

Not a useful error :(- ‘isTrusted’ means your browser allows indexedDB

IndexedDB API

The API is available through the global `indexedDB`

- Which holds an instance of IDBFactory

IDBFactory has (currently) four methods - only one which you should need:

open - useful to us

deleteDatabase - unlikely to be useful

- but could allow temporary databases to be deleted
- and help with backing up synchronising …

databases - unlikely to be useful

- allows you to list the databases - which we should know anyway ;)

cmp - I don’t expect you will need this …

See also https://developer.mozilla.org/en-US/docs/Web/API/IDBFactory

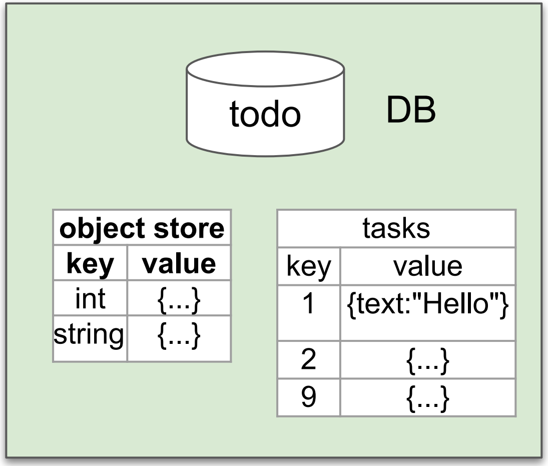
University of Sheffield

Object Stores ~~tables~~ - Todo list as example

Each database can hold multiple ‘object store(s)’

Similar to table/collection/document

- Stores data as objects (?!)
- Need a **unique** key (index)
- DB typically has several - <u>may have only one</u>



object store	
key	**value**
int	{...}
string	{...}

tasks	
key	value
1	{text:"Hello"}
2	{...}
9	{...}

Objects are stored using serialization and you MUST use transactions :|

- For our purposes, this is JSON.stringify

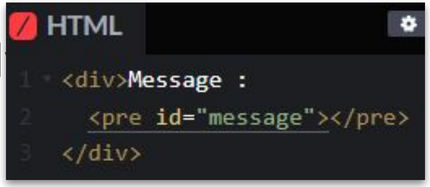
… and retrieved using serialization

- Equivalent to JSON.parse

Note that object serialization means functions are not serialised nor are prototype properties - you should be use empty javascript objects, i.e. {} - like Web Storage

Creating an object store (table)

Slight update to Html - just a tidy reall



```html
<div>Message :
   <pre id="message"></pre>
</div>
```

You need to:

- Developer tools >> delete database

Creating a database increase version from 0 (effectively) to 1 (or higher)

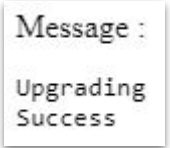
- **Have to handle ‘upgradeneeded’**
- *Open with no version means ‘1’*

Creating an object store (table)

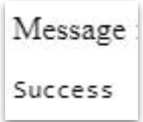
Have to handle ‘upgradeneeded’

- *version upgrades from (0) to 1*

1st time -> 2nd



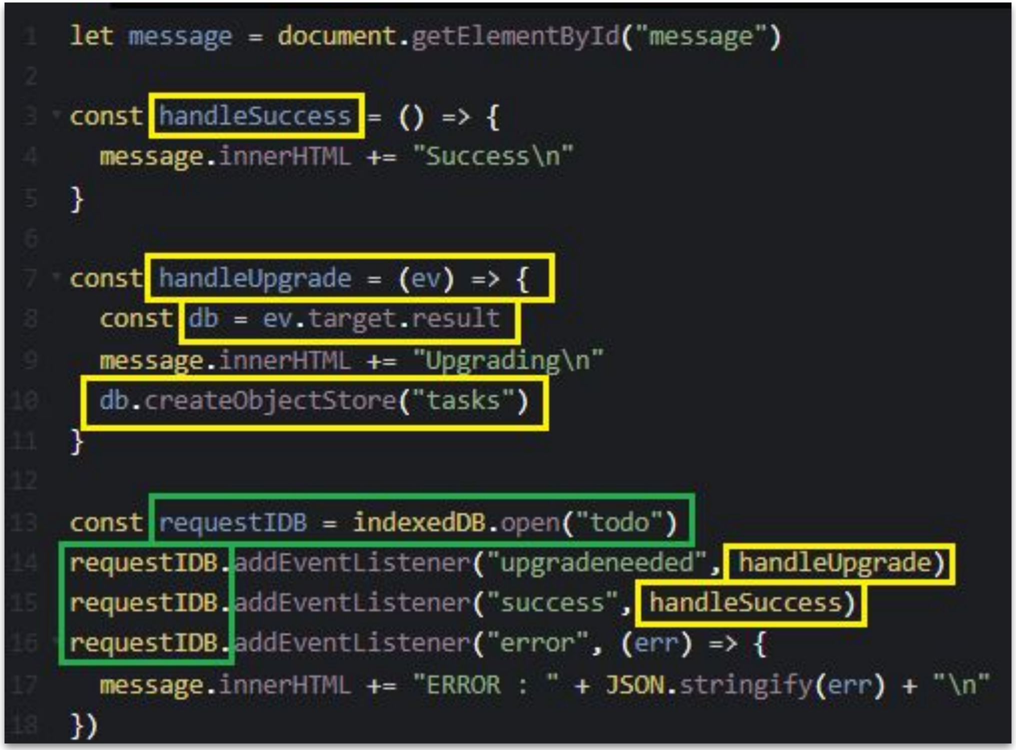
```
Message :

Upgrading
Success
```



```
Message :

Success
```

Notes:

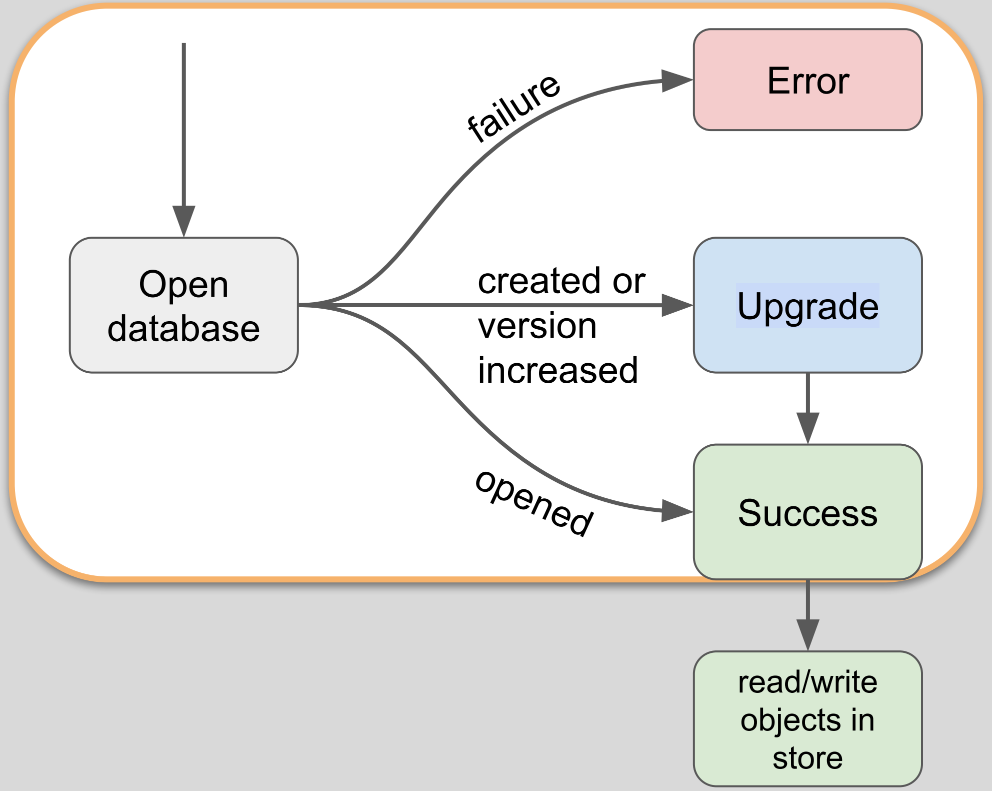
- *Will use the db object returned in the success handler*



```
let message = document.getElementById("message")

const handleSuccess = () => {
  message.innerHTML += "Success\n"
}

const handleUpgrade = (ev) => {
  const db = ev.target.result
  message.innerHTML += "Upgrading\n"
  db.createObjectStore("tasks")
}

const requestIDB = indexedDB.open("todo")
requestIDB.addEventListener("upgradeneeded", handleUpgrade)
requestIDB.addEventListener("success", handleSuccess)
requestIDB.addEventListener("error", (err) => {
  message.innerHTML += "ERROR : " + JSON.stringify(err) + "\n"
})
```

Reminder - Creating/opening Object Store

Remember - Browser JavaScript is single threaded (except for web workers)



E.g. code has older version, not trusted

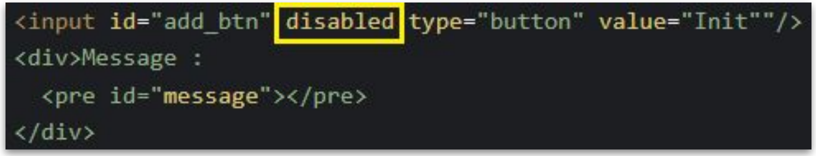
Can only add/modify Object Stores here
Also called on **<u>creation</u>**/first open
*N.B. version is an **integer**, i.e. 1.9 is 1*

Cannot use transactions before success

MUST read/write objects using transactions

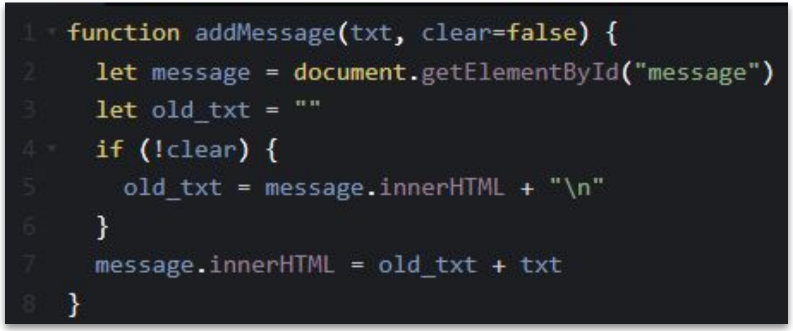
Writing/storing Objects 1

Update the Html to have a (initially disabled) ‘Init’ button:



```
<input id="add_btn" disabled type="button" value="Init""/>
<div>Message :
  <pre id="message"></pre>
</div>
```

Refactor by adding common message showing (assume this from now on):



```
function addMessage(txt, clear=false) {
  let message = document.getElementById("message")
  let old_txt = ""
  if (!clear) {
    old_txt = message.innerHTML + "\n"
  }
  message.innerHTML = old_txt + txt
}
```

This is a function - as is const addMessage = (...) =>

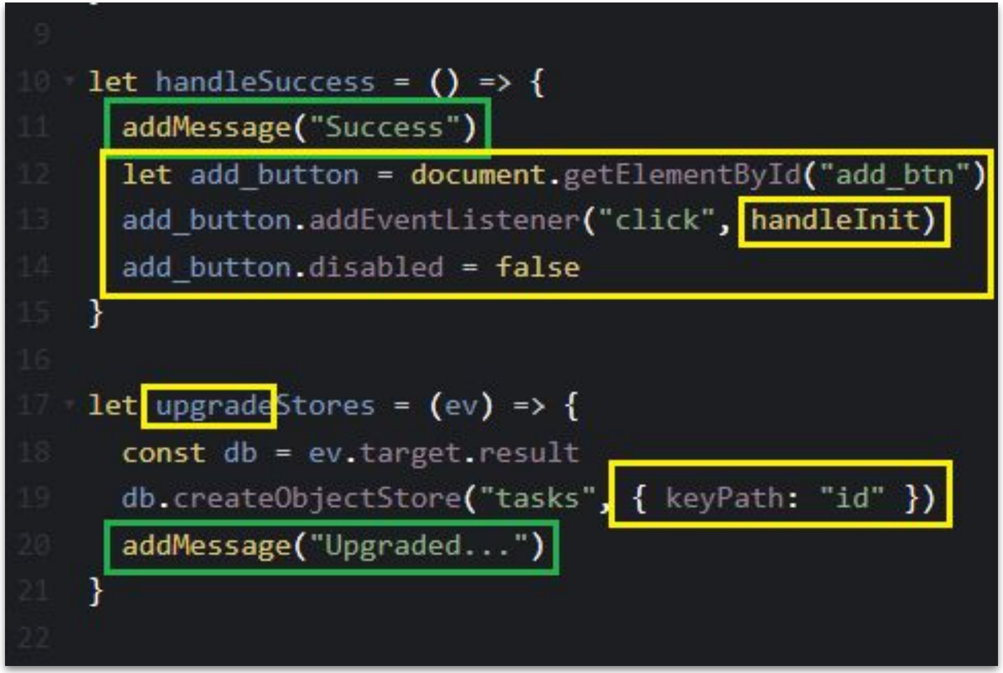
Writing Objects 2

You can use function if you prefer

- *There are reason/s to use const …*

Note:

- Using addMessage() …
- Success enables button
  - … and adds click handler (next slide)
- Upgrade now sets the expected unique key
- let used here instead of const
  - what is the difference?



```
let handleSuccess = () => {
  addMessage("Success")
  let add_button = document.getElementById("add_btn")
  add_button.addEventListener("click", handleInit)
  add_button.disabled = false
}

let upgradeStores = (ev) => {
  const db = ev.target.result
  db.createObjectStore("tasks", { keyPath: "id" })
  addMessage("Upgraded...")
}
```

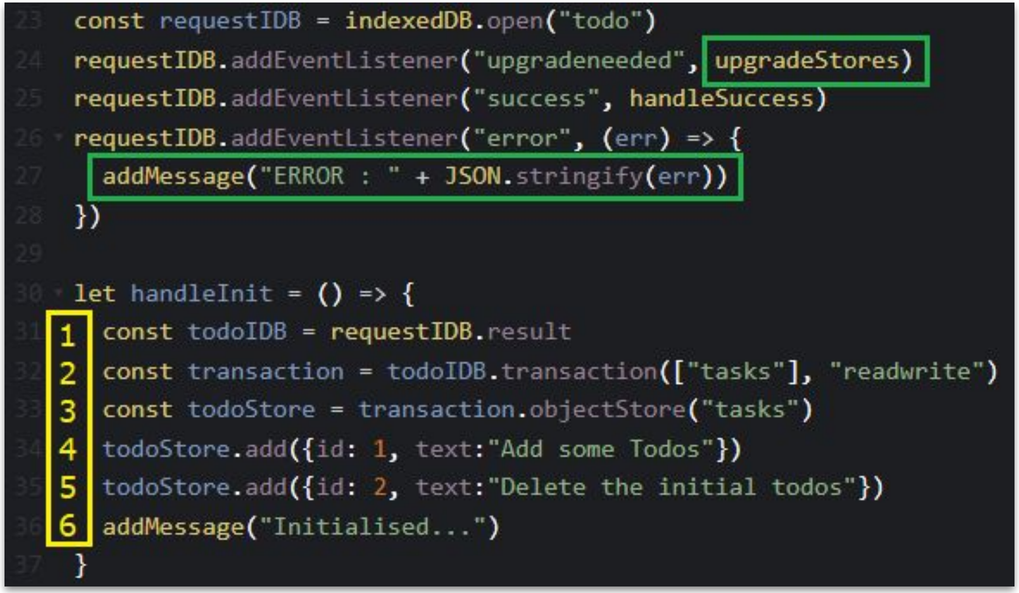
Storing Objects 3

The most important changes are:

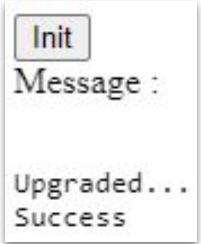
1. Get the IDBDatabase
2. All reads/writes to object stores **must** use transaction/s
    - Note the array. May pass “readonly”
3. The object store with transaction
4. Add an object. Must have `id` key
    - *Note ‘text’ is not in a ‘schema’*
5. *Add another object*
6. Show completion message …

Delete db then run

(Init will start disabled)



```
const requestIDB = indexedDB.open("todo")
requestIDB.addEventListener("upgradeneeded", upgradeStores)
requestIDB.addEventListener("success", handleSuccess)
requestIDB.addEventListener("error", (err) => {
  addMessage("ERROR : " + JSON.stringify(err))
})

let handleInit = () => {
1  const todoIDB = requestIDB.result
2  const transaction = todoIDB.transaction(["tasks"], "readwrite")
3  const todoStore = transaction.objectStore("tasks")
4  todoStore.add({id: 1, text:"Add some Todos"})
5  todoStore.add({id: 2, text:"Delete the initial todos"})
6  addMessage("Initialised...")
}
```



```
Message :


Upgraded...
Success
```

Transactions note

Transactions allow us to (just like with SQL):

- Combine multiple read/write/s safely to a database
- If there are any failures/errors, to ‘rollback’ so **no updates** are applied
- To ‘commit’ when the transaction is complete

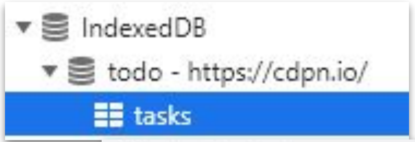
Transactions **must be** used to read/write from object stores

To start a transaction, we need to know:

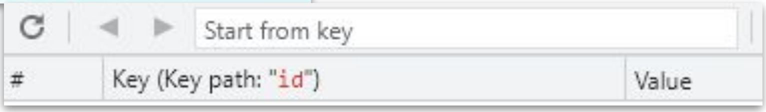
1. The object store(s) to ‘lock’
2. Whether they are locked for
    - readonly (default-better performance)
    - or readwrite

Storing Objects 4

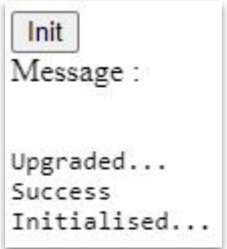
In developer tools, open 'tasks':



The (known) structure is shown:

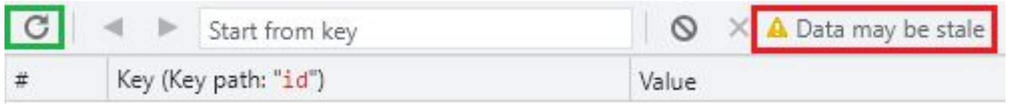


#	Key (Key path: "id")	Value



```
Upgraded...
Success
Initialised...
```

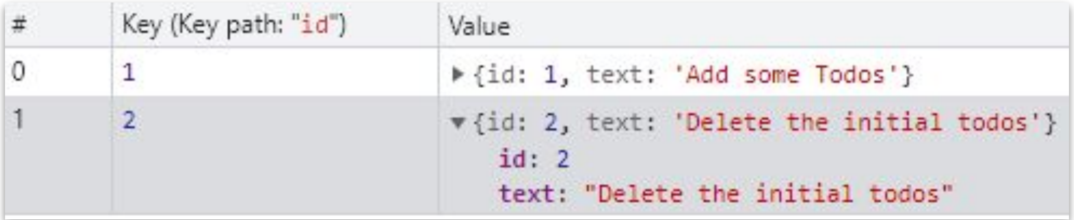
Click Init …



#	Key (Key path: "id")	Value

Refresh ⟳ to show:

- Shown with Value expanded…



#	Key (Key path: "id")	Value
0	1	▸{id: 1, text: 'Add some Todos'}
1	2	▾{id: 2, text: 'Delete the initial todos'} id: 2 text: "Delete the initial todos"

N.B. There is no schema …

IDBDatabase

The database is returned from open as **IDBDatabase**

Most useful methods:

- transaction()
- createObjectStore() - only when handling version change in upgradeneeded

May be useful:

- objectStoreNames - a list of …
- version - integer version >0
- close() - rarely needed
- deleteObjectStore() - useful for temporary object stores

See also https://developer.mozilla.org/en-US/docs/Web/API/IDBDatabase

Transactions must FULLY complete

Transactions force every db read/write to succeed for the actions to persist.

Try this in the lab:

1. Delete the first object (“Add some todos”), leaving the second one
2. Now try ‘Init’ and refresh the shown data - there is no change
   - When the first write failed - due to same id - the transaction failed
3. Delete the second object and then ‘Init’ - both are now added
4. Now delete the second and then init - again - none are added
   - The first write succeeded, but the second failed, so the transaction failed to commit

Short break

What happens if you declare a global function xxx() {... } twice?

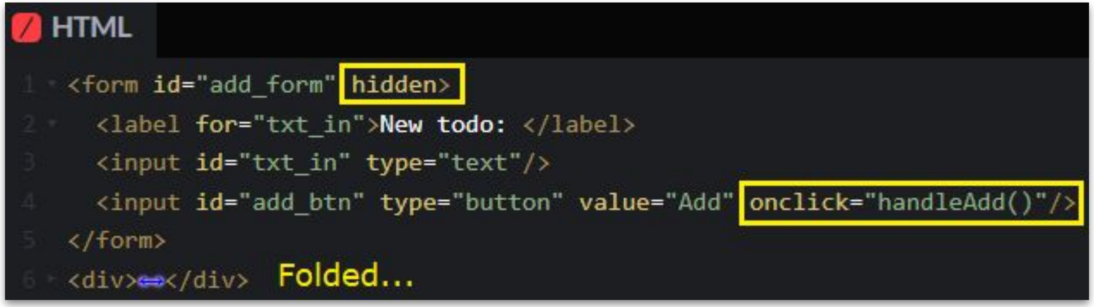
The last one replaces the earlier one with no warning :(
unlike const xxx = () => { … }

N.B. This is one reason const is good :)

Adding new Todo

Updating Html to add input:

*N.B. I've shown the click handler linked here - but there are reasons we shouldn't do this … **avoid this :(***



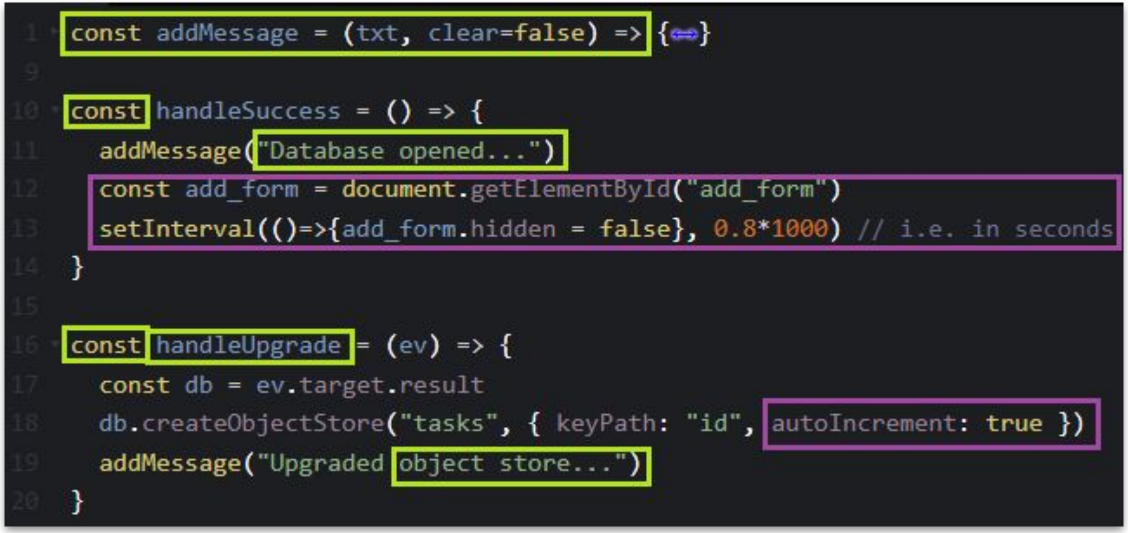
HTML

```
<form id="add_form" hidden>
  <label for="txt_in">New todo: </label>
  <input id="txt_in" type="text"/>
  <input id="add_btn" type="button" value="Add" onclick="handleAdd()"/>
</form>
<div>↔</div>  Folded...
```

- + const gives some protection
- + added autoIncrement
- + reveal form (after 0.8 secs)

Note that (again) success enables interaction …

Note: const can't be 'clobbered' - unlike function

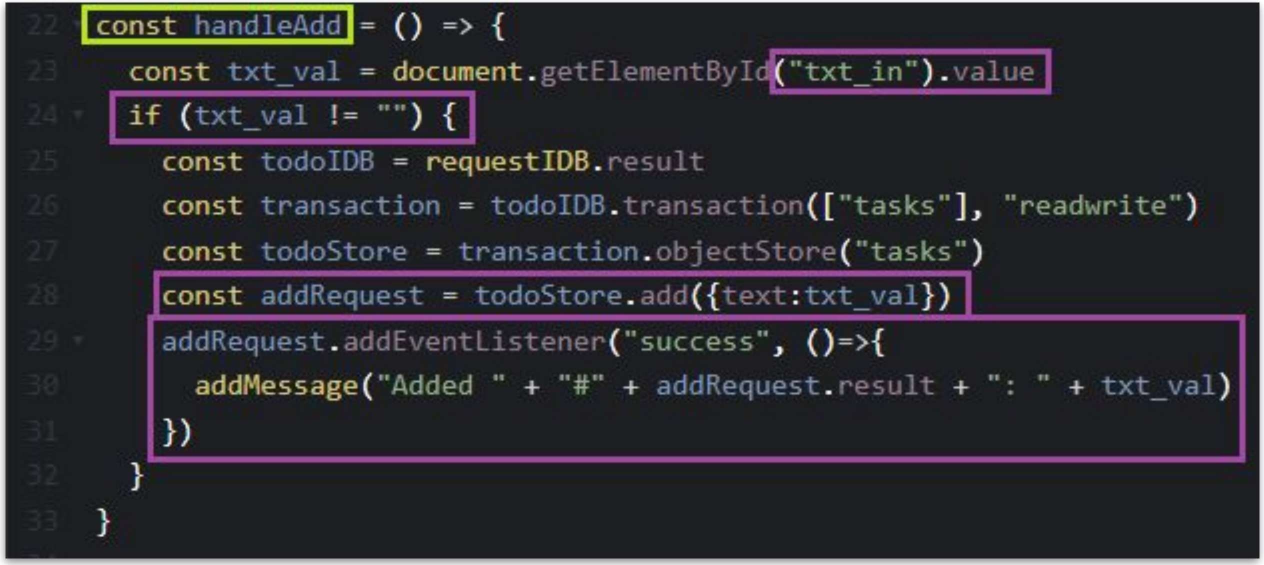


```
const addMessage = (txt, clear=false) => {↔}

const handleSuccess = () => {
  addMessage("Database opened...")
  const add_form = document.getElementById("add_form")
  setInterval(()=>{add_form.hidden = false}, 0.8*1000) // i.e. in seconds
}

const handleUpgrade = (ev) => {
  const db = ev.target.result
  db.createObjectStore("tasks", { keyPath: "id", autoIncrement: true })
  addMessage("Upgraded object store...")
}
```

Adding user's todo - part 2

handleAdd replaces (and extends) the Init handler:

Note that `<store>.add` no longer passes in the id - autoincrement = true



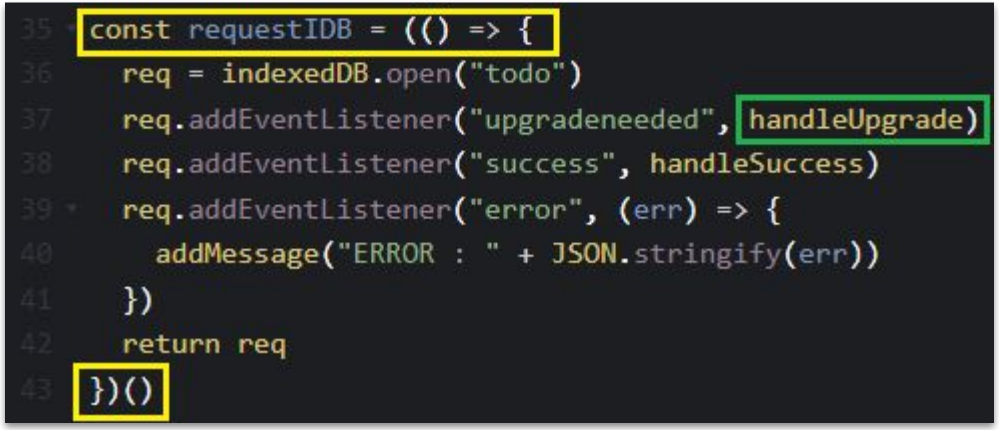
```
const handleAdd = () => {
  const txt_val = document.getElementById("txt_in").value
  if (txt_val != "") {
    const todoIDB = requestIDB.result
    const transaction = todoIDB.transaction(["tasks"], "readwrite")
    const todoStore = transaction.objectStore("tasks")
    const addRequest = todoStore.add({text:txt_val})
    addRequest.addEventListener("success", ()=>{
      addMessage("Added " + "#" + addRequest.result + ": " + txt_val)
    })
  }
}
```

Also, there is a success listener on the add request - this will show the inserted id

Adding user's todo - part 3

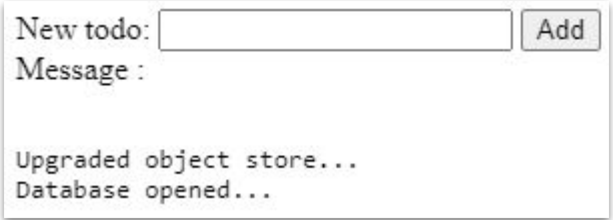
Finally - this is an IIFE (dated approach)
Immediately Invoked Function Expression

- creates an anonymous function
- then makes it an (expression)
- then invokes it as (expression)()
- this also returns it's result to global const requestIDB



```
const requestIDB = (() => {
  req = indexedDB.open("todo")
  req.addEventListener("upgradeneeded", handleUpgrade)
  req.addEventListener("success", handleSuccess)
  req.addEventListener("error", (err) => {
    addMessage("ERROR : " + JSON.stringify(err))
  })
  return req
})()
```

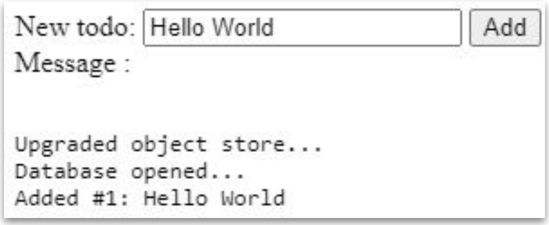
Delete the database then run …



New todo: [ ] Add
Message :

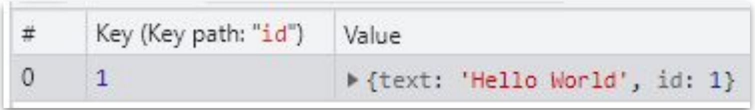
```
Upgraded object store...
Database opened...
```

Add "Hello World"



New todo: Hello World Add
Message :

```
Upgraded object store...
Database opened...
Added #1: Hello World
```



#	Key (Key path: "id")	Value
0	1	▸{text: 'Hello World', id: 1}

Design issues / backlog

These include:

- How to do ~~C~~RUD (create read update delete)
  - show todos
  - Sort re/order (update)
  - delete
  - edit (update)
- and (later) GUI improvements …
- Also - fix the global cloberring

Doing these in sprints:
1. Fix global clobber
2. How to show todos
3. insert Sorted
4. Update todo

Sprint 1 - fix global clober

Could us an IIFE

ES6 better to use {scope}

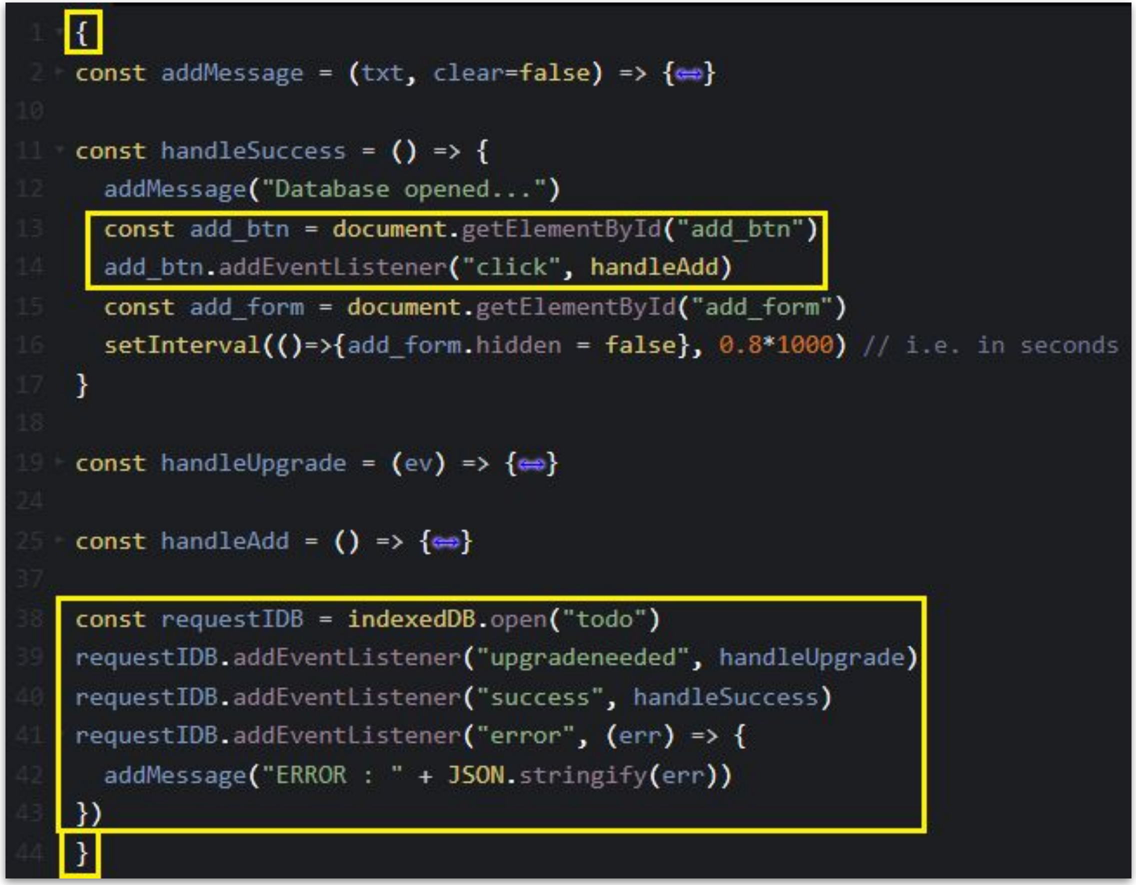
- fork the Pen
- must also fix onclick:

```html
<input id="add_btn" type="button" value="Add"/>
```

Note the { … }

Also setup is now just code (?)

const is good practice



```javascript
{
const addMessage = (txt, clear=false) => {⇔}

const handleSuccess = () => {
  addMessage("Database opened...")
  const add_btn = document.getElementById("add_btn")
  add_btn.addEventListener("click", handleAdd)
  const add_form = document.getElementById("add_form")
  setInterval(()=>{add_form.hidden = false}, 0.8*1000) // i.e. in seconds
}

const handleUpgrade = (ev) => {⇔}

const handleAdd = () => {⇔}

const requestIDB = indexedDB.open("todo")
requestIDB.addEventListener("upgradeneeded", handleUpgrade)
requestIDB.addEventListener("success", handleSuccess)
requestIDB.addEventListener("error", (err) => {
  addMessage("ERROR : " + JSON.stringify(err))
})
}
```

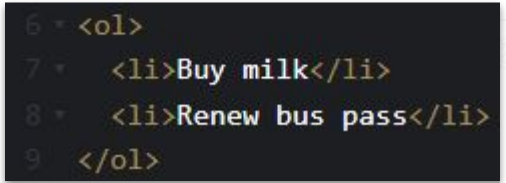
University of Sheffield

Sprint 2 - How to show todos

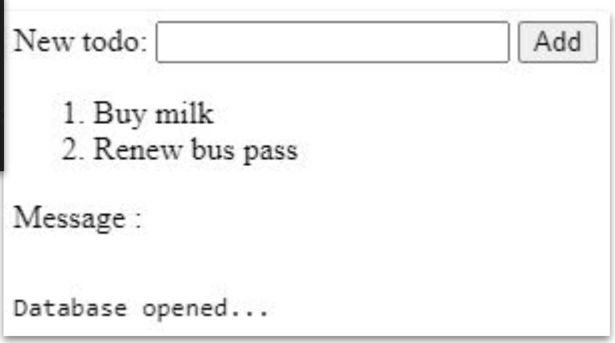
We need to design this roughly

- here's a 'back of an envelope' design
- …actually a receipt
  - +++quick
  - +++cheap
  - +++easily changed

Then update html for a 'realistic data' example:



```
<ol>
  <li>Buy milk</li>
  <li>Renew bus pass</li>
</ol>
```

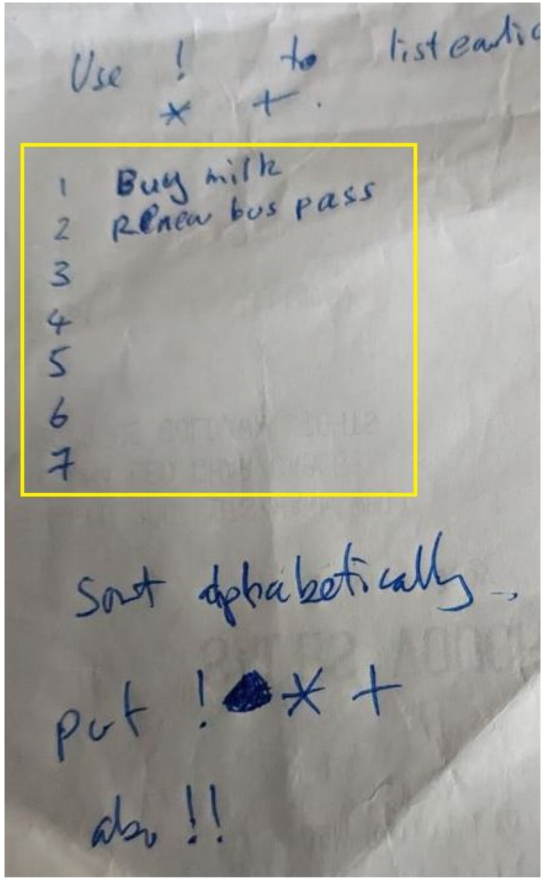


New todo: [ ] Add

1. Buy milk
2. Renew bus pass

Message :

```
Database opened...
```



Use ! * + to list earlic

1 Buy milk
2 Renew bus pass
3
4
5
6
7

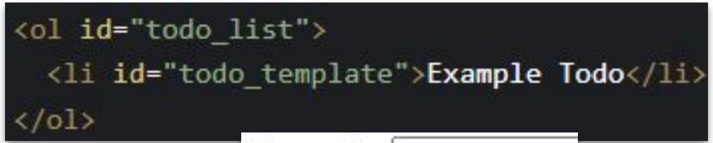
Sort alphabetically →

put ! * +

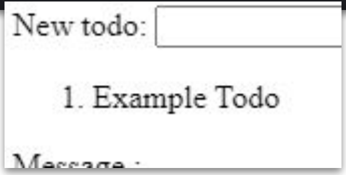
also !!

Sprint 2 - use a template item

Create an example to copy:



```
<ol id="todo_list">
  <li id="todo_template">Example Todo</li>
</ol>
```

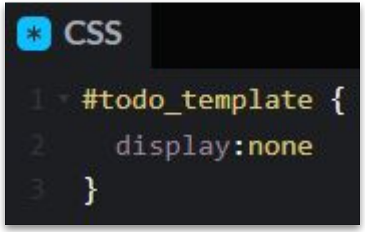


New todo:

1. Example Todo

Message :

and hide using CSS it at start?!



CSS

```
#todo_template {
   display:none
 }
```

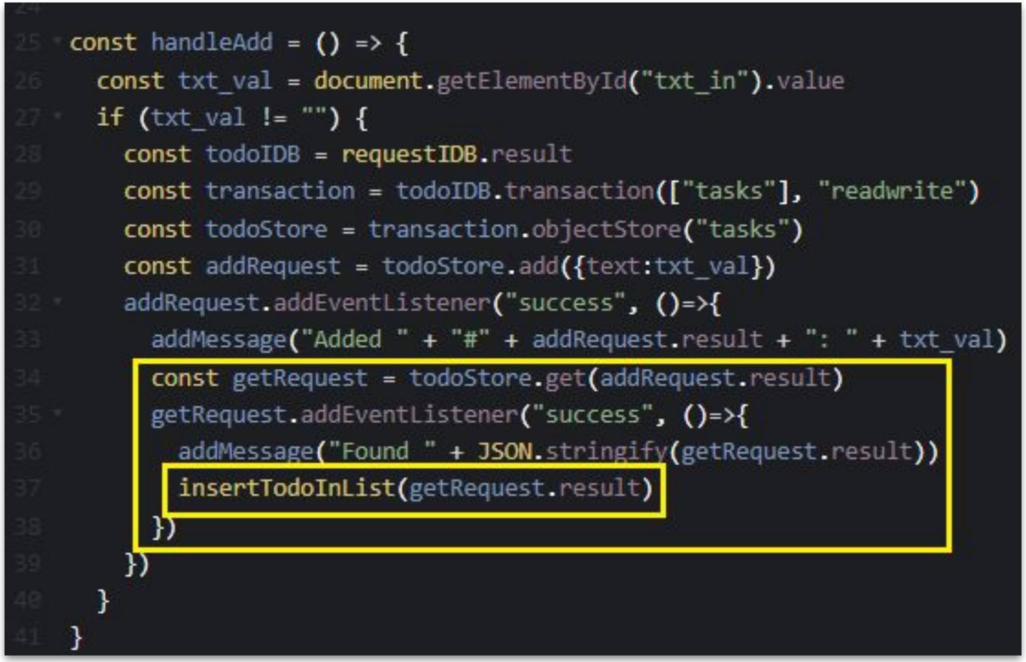
University of Sheffield

Sprint 2 - add to the database

Remember - the Add button click will call handleAdd

Notes:

- we need to inserted object id
- using get
  - with the add result
- retrieve the todo object
- call insertTodoInList
  - see following…

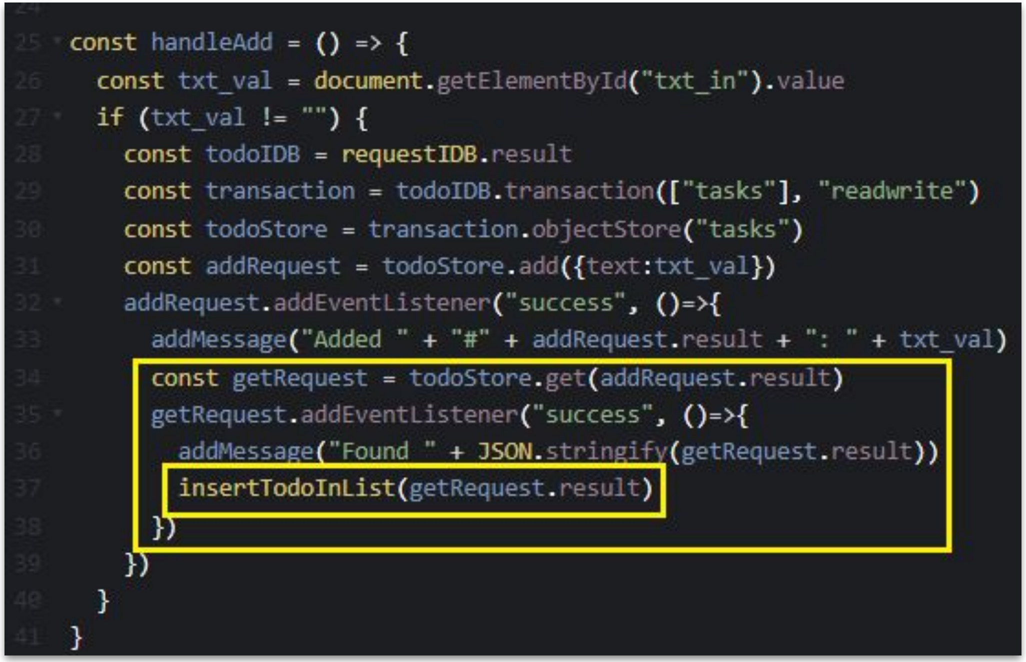


```
const handleAdd = () => {
  const txt_val = document.getElementById("txt_in").value
  if (txt_val != "") {
    const todoIDB = requestIDB.result
    const transaction = todoIDB.transaction(["tasks"], "readwrite")
    const todoStore = transaction.objectStore("tasks")
    const addRequest = todoStore.add({text:txt_val})
    addRequest.addEventListener("success", ()=>{
      addMessage("Added " + "#" + addRequest.result + ": " + txt_val)
      const getRequest = todoStore.get(addRequest.result)
      getRequest.addEventListener("success", ()=>{
        addMessage("Found " + JSON.stringify(getRequest.result))
        insertTodoInList(getRequest.result)
      })
    })
  }
}
```

Example Transaction detail

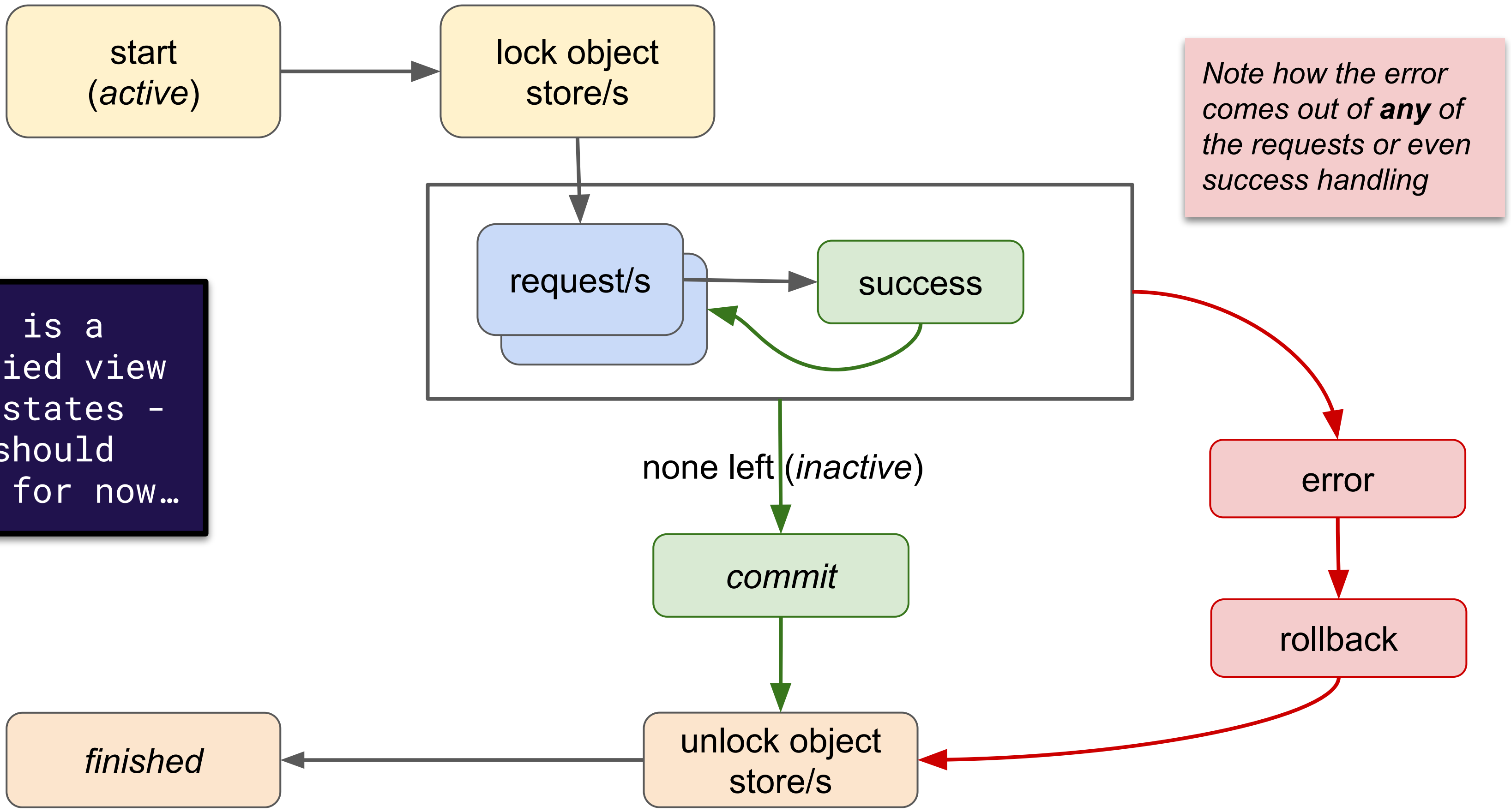
In the previous example - we have:

- A request (to add) is made within the transaction
- The **add success** is handled, and a get is requested within the same transaction
- The **get request** success is then handled
    - still within the transaction
- When there are no more transaction tasks (request/s) then the transaction will automatically be committed



```
const handleAdd = () => {
  const txt_val = document.getElementById("txt_in").value
  if (txt_val != "") {
    const todoIDB = requestIDB.result
    const transaction = todoIDB.transaction(["tasks"], "readwrite")
    const todoStore = transaction.objectStore("tasks")
    const addRequest = todoStore.add({text:txt_val})
    addRequest.addEventListener("success", ()=>{
      addMessage("Added " + "#" + addRequest.result + ": " + txt_val)
      const getRequest = todoStore.get(addRequest.result)
      getRequest.addEventListener("success", ()=>{
        addMessage("Found " + JSON.stringify(getRequest.result))
        insertTodoInList(getRequest.result)
      })
    })
  }
}
```

Overview of Transaction states



*Note how the error comes out of **any** of the requests or even success handling*

This is a simplified view of the states - but should suffice for now…

Transactions and async

N.B. Transaction/s automatically commit - when there are no more requests

- Warning - this means we can’t do **any** async operations **during a transaction**

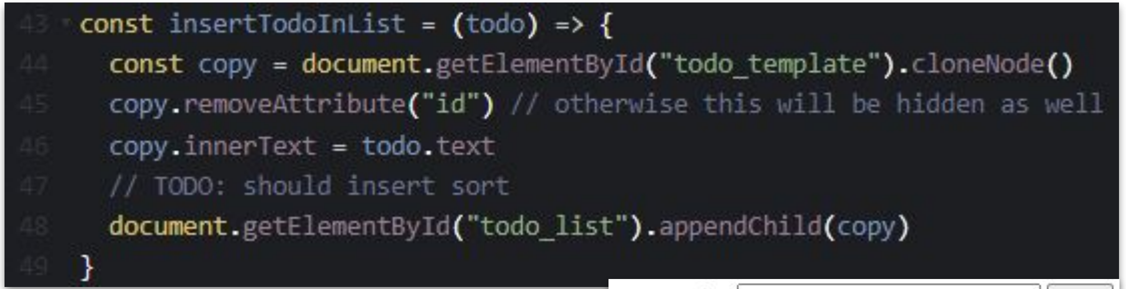
i.e. DURING A TRANSACTION - DO NOT USE

  - fetch
  - setTimeout, setInterval

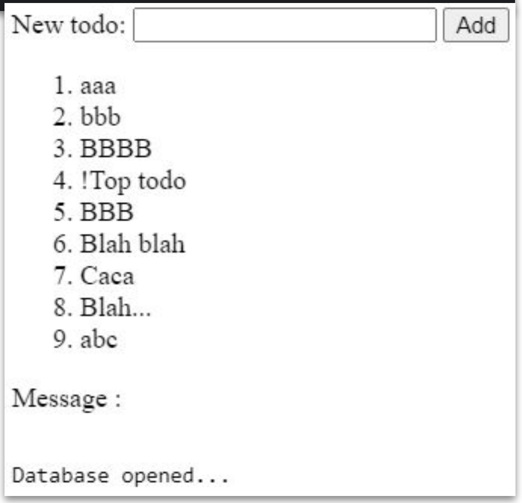
Sprint 2 - Clone template item

Note:

- This takes a todo object
- clone the template element
- remove the **DOM** id
- set the text
- Then append
  - Incorrectly - should insert sort



```
const insertTodoInList = (todo) => {
  const copy = document.getElementById("todo_template").cloneNode()
  copy.removeAttribute("id") // otherwise this will be hidden as well
  copy.innerText = todo.text
  // TODO: should insert sort
  document.getElementById("todo_list").appendChild(copy)
}
```



1. aaa
2. bbb
3. BBBB
4. !Top todo
5. BBB
6. Blah blah
7. Caca
8. Blah...
9. abc

Message :

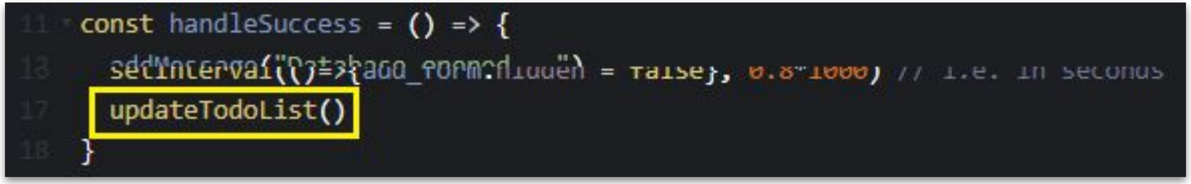
```
Database opened...
```

Also need to retrieve all the objects on open and insert them …

- Where should this be done?

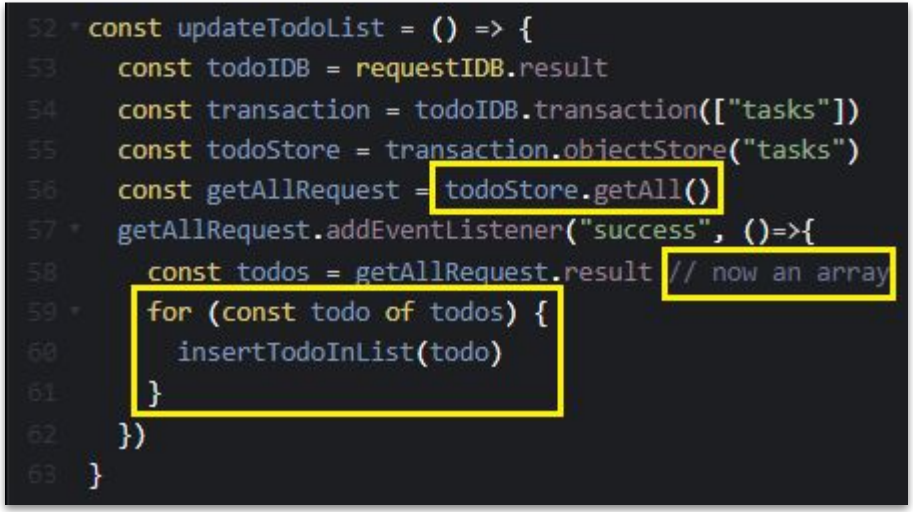
Sprint 2 - Insert all todos

Answer - at end of object store open success



```
const handleSuccess = () => {
  addMessage("Database opened")
  setInterval(()=>{add_form.hidden = false}, 0.8*1000) // i.e. in seconds
  updateTodoList()
}
```

Inserting all follows same approach as get, etc.



```
const updateTodoList = () => {
  const todoIDB = requestIDB.result
  const transaction = todoIDB.transaction(["tasks"])
  const todoStore = transaction.objectStore("tasks")
  const getAllRequest = todoStore.getAll()
  getAllRequest.addEventListener("success", ()=>{
    const todos = getAllRequest.result // now an array
    for (const todo of todos) {
      insertTodoInList(todo)
    }
  })
}
```

Aside - Single Page Application (SPA)

Note how this is a Single Page Application:

- The page is loaded once
- The content is updated without reloading
- We currently don’t access anything on the server - but if we did:
  - The application would use JSON and REST (typically) with DOM (Document Object Model) updates
  - Using fetch/Ajax
- Currently this works as an offline SPA
  - If the internet is down, the page still works
  - Except if we force a reload - we will fix this later

We will return to SPAs …

Short break

How do you cope with technical debt?

Use refactoring at the end of each sprint (at least).
This should fit with tests passing - see Red Green Refactor

Sprint 3 - insert Sorted

One option would be to use an element (node) id attribute

- BUT - these could clash with other id's, so instead we use a '**data-**set'

Data set attributes called '**data-**XXX' - in our case '**data-**todo-id'

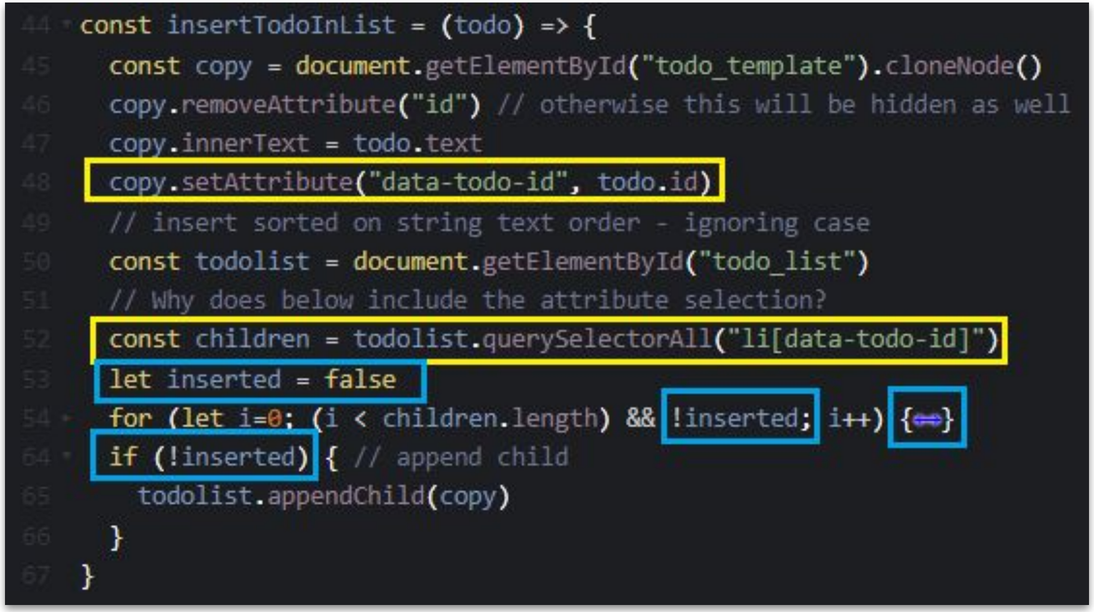
Note: We could just reread all the todos after a successful add (instead of inserting)

*But that is an expensive (energy and time) thing to do … looks like a '**code smell'***

Sprint 3 - update insertTodoInList

Notes:

- data-todo-id attribute added
  - holds db id for later…
- also used to query(All)
  - using li[data-todo-id]
  - i.e. not template…
- ‘inserted’ keeps track of loop exit
  - and whether to append



```
const insertTodoInList = (todo) => {
  const copy = document.getElementById("todo_template").cloneNode()
  copy.removeAttribute("id") // otherwise this will be hidden as well
  copy.innerText = todo.text
  copy.setAttribute("data-todo-id", todo.id)
  // insert sorted on string text order - ignoring case
  const todolist = document.getElementById("todo_list")
  // Why does below include the attribute selection?
  const children = todolist.querySelectorAll("li[data-todo-id]")
  let inserted = false
  for (let i=0; (i < children.length) && !inserted; i++) {⇔}
  if (!inserted) { // append child
    todolist.appendChild(copy)
  }
}
```

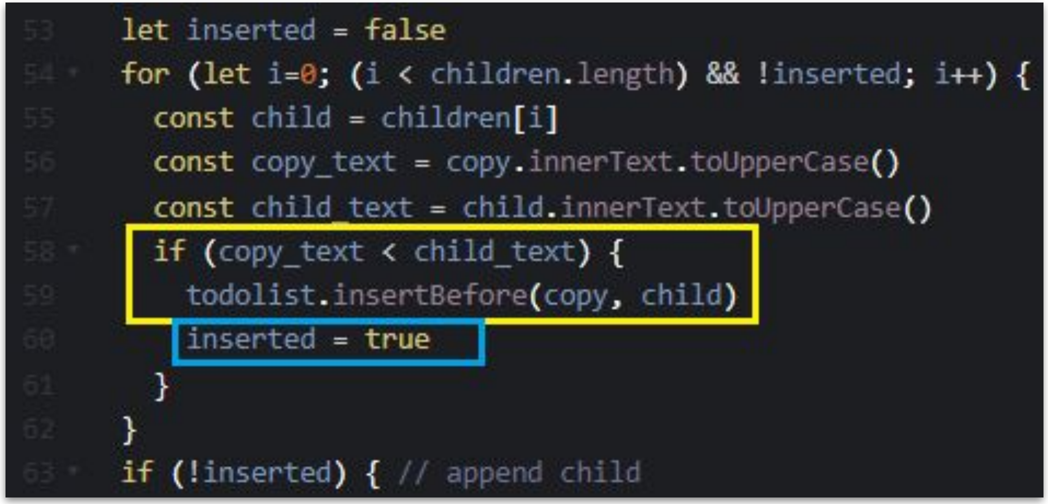
Folded due to size …

University of Sheffield

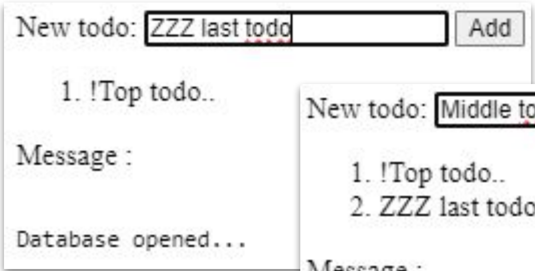
Sprint 3 - update insertTodoInList

Notes:

- *'inserted' is true to exit loop*
- using '<' inserts todo/s in creation order where same text
  - <= would insert newest first - which currently would have no visible effect



```
let inserted = false
for (let i=0; (i < children.length) && !inserted; i++) {
  const child = children[i]
  const copy_text = copy.innerText.toUpperCase()
  const child_text = child.innerText.toUpperCase()
  if (copy_text < child_text) {
    todolist.insertBefore(copy, child)
    inserted = true
  }
}
if (!inserted) { // append child
```

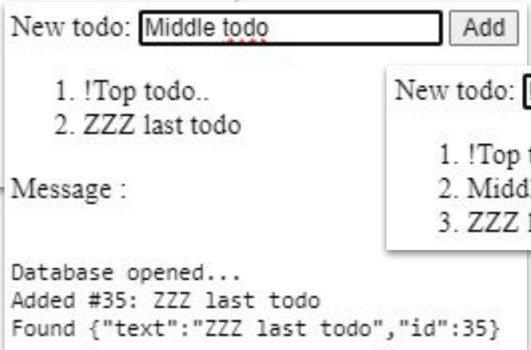


New todo: ZZZ last todo Add

1. !Top todo..

Message :

```
Database opened...
```

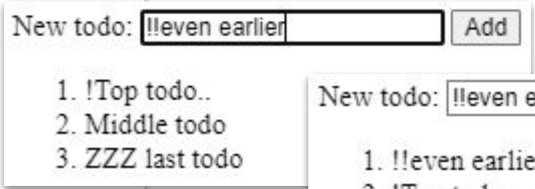


New todo: Middle todo Add

1. !Top todo..
2. ZZZ last todo

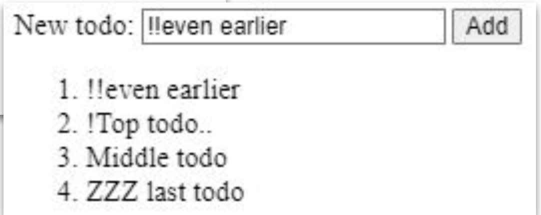
Message :

```
Database opened...
Added #35: ZZZ last todo
Found {"text":"ZZZ last todo","id":35}
```



New todo: !!even earlier Add

1. !Top todo..
2. Middle todo
3. ZZZ last todo



New todo: !!even earlier Add

1. !!even earlier
2. !Top todo..
3. Middle todo
4. ZZZ last todo

This is one way to do this - in vanilla JS

Sprint 4 - update Todo

Designing so the user can:

- Click on an existing todo item
- The text will be copied into the input text box
- The user can change what it says …
- And then click update to save the changes

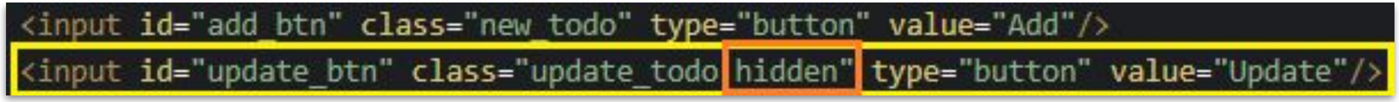
Note: We need to keep track of the data id

Note how sprints allow us to move forward in focused ‘steps’

Sprint 4 - Update todo

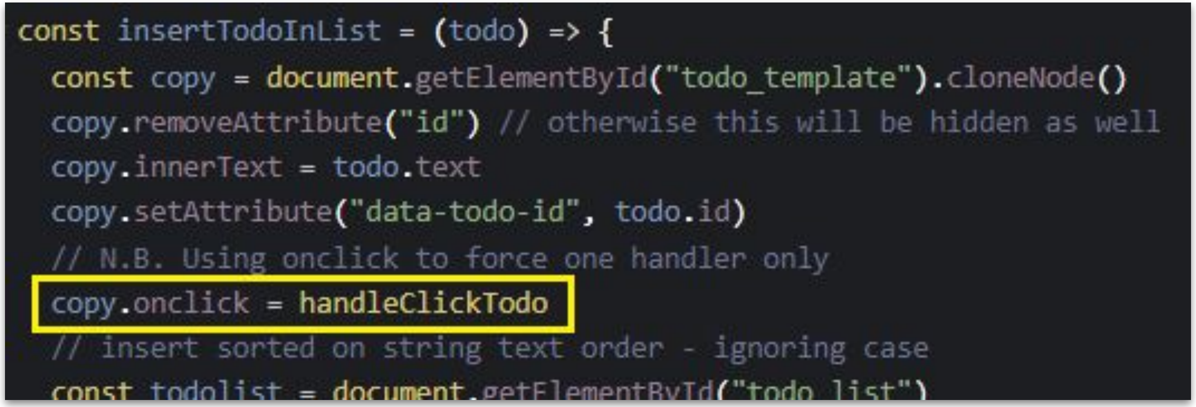
Need:

- an update/save button (hidden at start and when not editing)



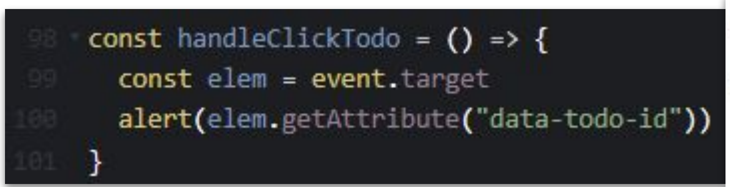
```html
<input id="add_btn" class="new_todo" type="button" value="Add"/>
<input id="update_btn" class="update_todo hidden" type="button" value="Update"/>
```

- to allow click on (any) todo item:

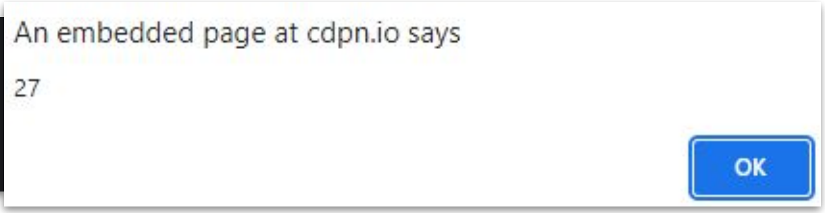


```javascript
const insertTodoInList = (todo) => {
  const copy = document.getElementById("todo_template").cloneNode()
  copy.removeAttribute("id") // otherwise this will be hidden as well
  copy.innerText = todo.text
  copy.setAttribute("data-todo-id", todo.id)
  // N.B. Using onclick to force one handler only
  copy.onclick = handleClickTodo
  // insert sorted on string text order - ignoring case
  const todolist = document.getElementById("todo_list")
```

  - Test this now - should show alert:



```javascript
const handleClickTodo = () => {
  const elem = event.target
  alert(elem.getAttribute("data-todo-id"))
}
```



An embedded page at cdpn.io says

27

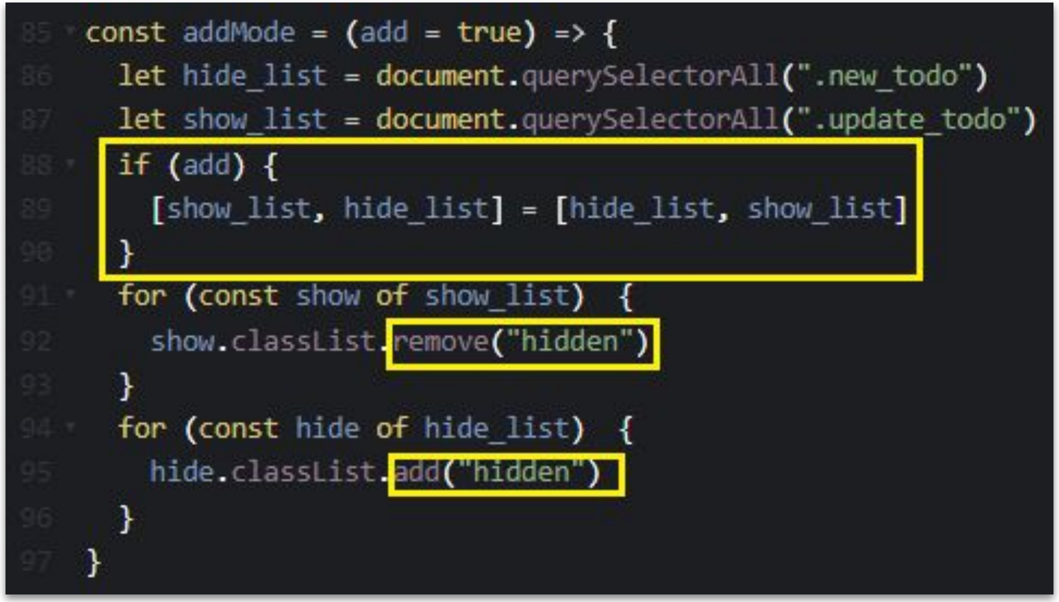
Sprint 4 - Update visibility of buttons

- Add a function to toggle visibility between add and update
  - Based on element class

Notes:

- We start assuming update
  - i.e. the hidelist and show list are for (add == false)
- Then swap lists if add is true
  - or parameter not passed?!
- Note how each list has the hidden class added/removed

This is a useful pattern for vanilla JS and the assignment :)

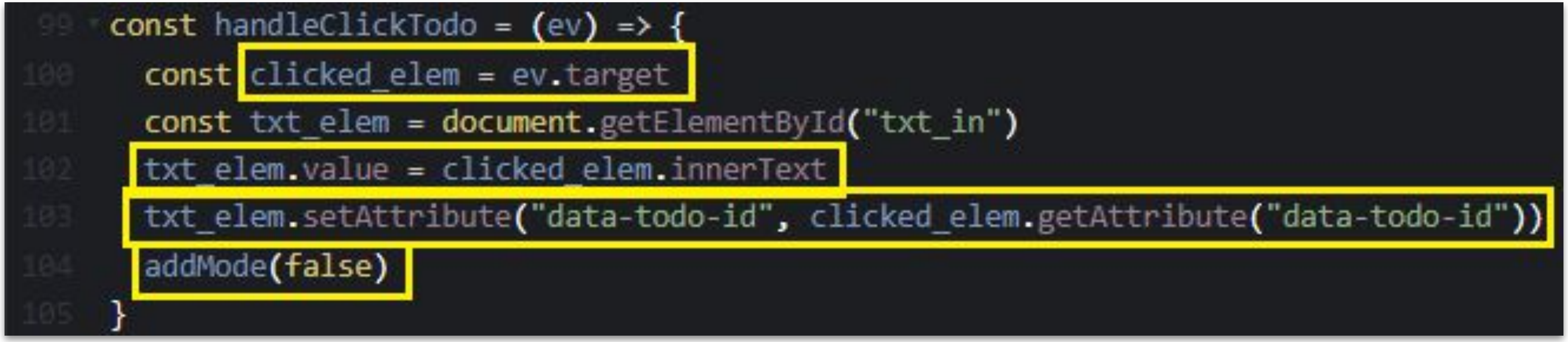


```
const addMode = (add = true) => {
  let hide_list = document.querySelectorAll(".new_todo")
  let show_list = document.querySelectorAll(".update_todo")
  if (add) {
    [show_list, hide_list] = [hide_list, show_list]
  }
  for (const show of show_list)  {
    show.classList.remove("hidden")
  }
  for (const hide of hide_list)  {
    hide.classList.add("hidden")
  }
}
```

Sprint 4 - Handle click on a todo

A few things need to happen:

- Get the clicked element - ev.target
- Transfer the text/description



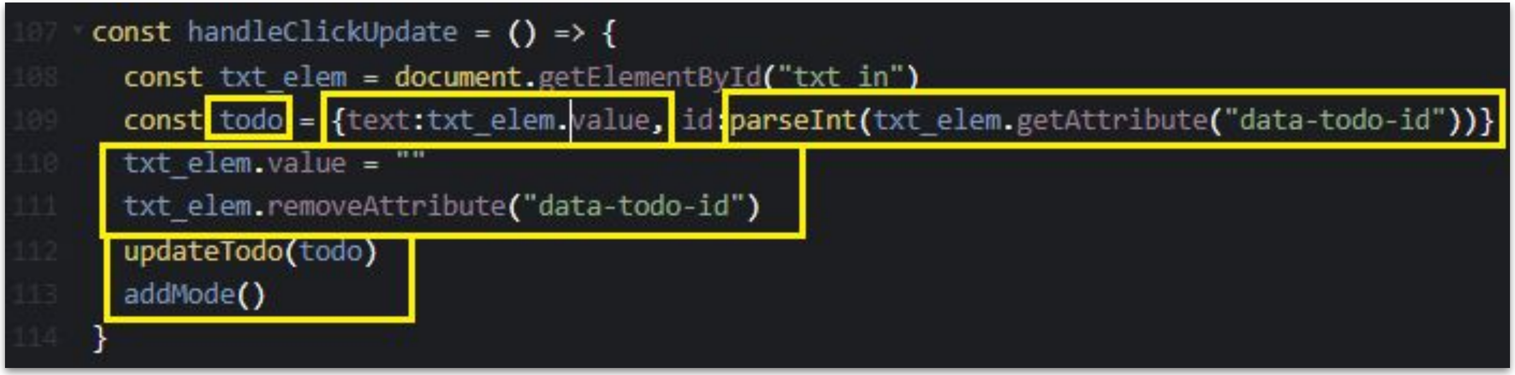
```
const handleClickTodo = (ev) => {
  const clicked_elem = ev.target
  const txt_elem = document.getElementById("txt_in")
  txt_elem.value = clicked_elem.innerText
  txt_elem.setAttribute("data-todo-id", clicked_elem.getAttribute("data-todo-id"))
  addMode(false)
}
```

- Store the indexeddb id
- Switch the button visibility - i.e. addMode becomes false

Sprint 4 - Handle click on Update

When the user clicks ‘update’, need to:

- Get the (new) description
- Get the id
- Clear the entered text (and remove the id)
- Create a (simple) Todo object
- Update the todo - See next …
- Update the update/add button visibility

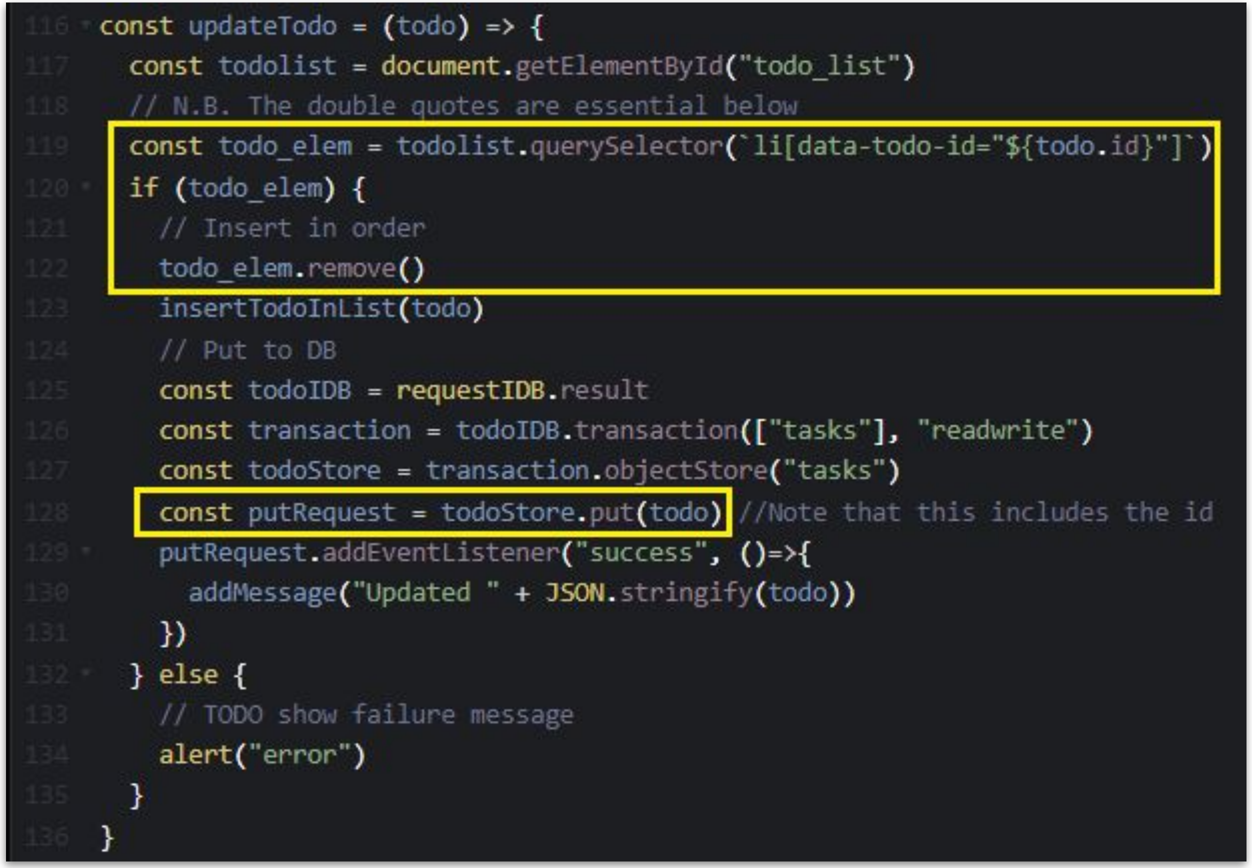


```
const handleClickUpdate = () => {
  const txt_elem = document.getElementById("txt_in")
  const todo = {text:txt_elem.value, id:parseInt(txt_elem.getAttribute("data-todo-id"))}
  txt_elem.value = ""
  txt_elem.removeAttribute("data-todo-id")
  updateTodo(todo)
  addMode()
}
```

Sprint 4 - Update the todo

Need to:

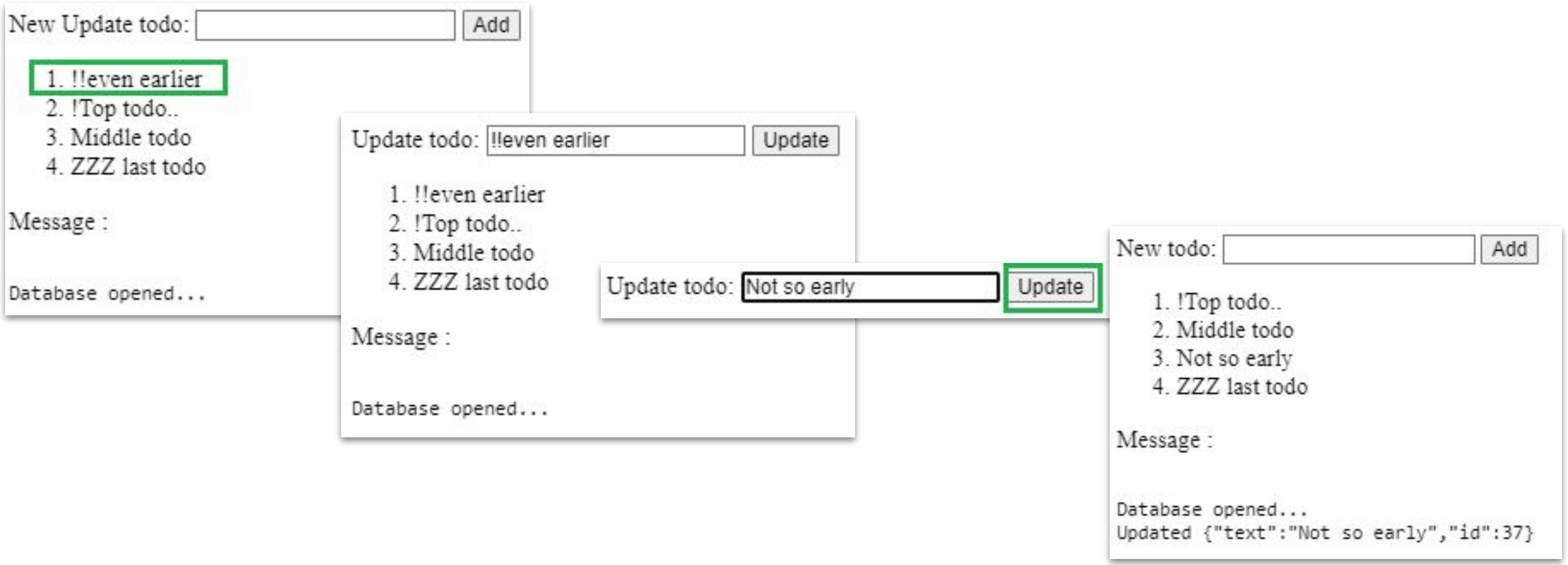
- Delete previous todo element
- Insert (as before)
  - Note: Since browser javascript is single threaded, the previous two actions are effectively 'atomic'
- Update the indexeddb <u>using put with the id</u> (not add)



```
const updateTodo = (todo) => {
  const todolist = document.getElementById("todo_list")
  // N.B. The double quotes are essential below
  const todo_elem = todolist.querySelector(`li[data-todo-id="${todo.id}"]`)
  if (todo_elem) {
    // Insert in order
    todo_elem.remove()
    insertTodoInList(todo)
    // Put to DB
    const todoIDB = requestIDB.result
    const transaction = todoIDB.transaction(["tasks"], "readwrite")
    const todoStore = transaction.objectStore("tasks")
    const putRequest = todoStore.put(todo) //Note that this includes the id
    putRequest.addEventListener("success", ()=>{
      addMessage("Updated " + JSON.stringify(todo))
    })
  } else {
    // TODO show failure message
    alert("error")
  }
}
```

Sprint 4 - Update the todo

- Showing interaction:



New Update todo: [ ] Add

1. !!even earlier
2. !Top todo..
3. Middle todo
4. ZZZ last todo

Message :

```
Database opened...
```

Update todo: !!even earlier Update

1. !!even earlier
2. !Top todo..
3. Middle todo
4. ZZZ last todo

Message :

```
Database opened...
```

Update todo: Not so early Update

New todo: [ ] Add

1. !Top todo..
2. Middle todo
3. Not so early
4. ZZZ last todo

Message :

```
Database opened...
Updated {"text":"Not so early","id":37}
```

Deleting (persisted) Data

In General, especially in Enterprise/s, we don't delete from databases

- we mark as 'deleted/out of date'
- Why? Imagine deleting a user - what problems might this cause?
  - Their purchases and history aren't linked to a valid account
  - If you allocate a new user - they might now be given the 'orphaned' account id :<
  - Deletion might be by accident - or malicious
- So - we tend to add a 'suspended' state to the user account

However - indexedDB is different:

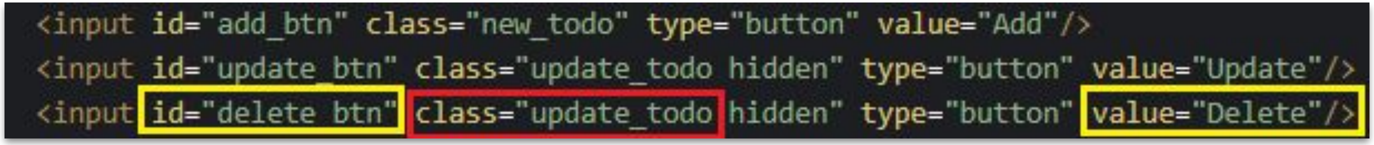
- We need to **minimize** memory usage
- The server DB (MongoDB) can keep history
- The server DB can always be used to restore …

Sprint 5 - Delete todo - design

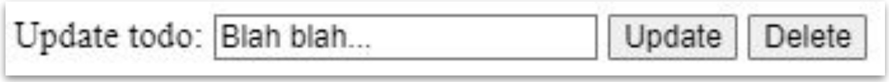
Where do we show delete?

For simplicity (and to allow a degree of confirmation):

- The user has to select (i.e. for updating) a todo to be able to delete it
- A Delete (button) will be needed
  - This will show an alert **confirm**ation to proceed



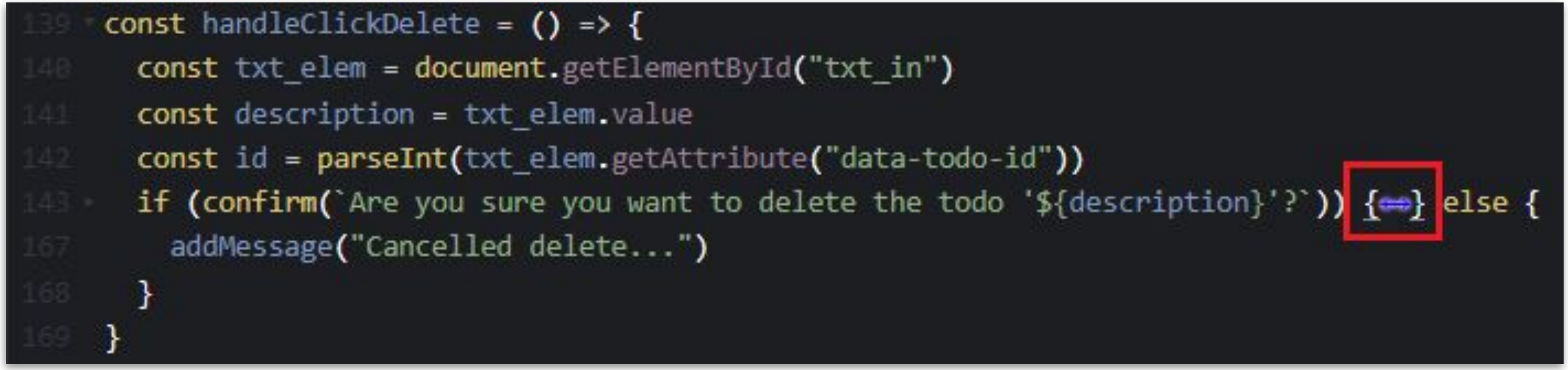
```
<input id="add_btn" class="new_todo" type="button" value="Add"/>
<input id="update_btn" class="update_todo hidden" type="button" value="Update"/>
<input id="delete_btn" class="update_todo hidden" type="button" value="Delete"/>
```



Remember to add the click handler to call - handleDeleteClick …

Sprint 5 - Delete a todo

This is a bit long (needs refactoring) so shown folded:



```
const handleClickDelete = () => {
  const txt_elem = document.getElementById("txt_in")
  const description = txt_elem.value
  const id = parseInt(txt_elem.getAttribute("data-todo-id"))
  if (confirm(`Are you sure you want to delete the todo '${description}'?`)) {⟺} else {
    addMessage("Cancelled delete...")
  }
}
```

Above shows a confirm modal dialog - with the todo text description

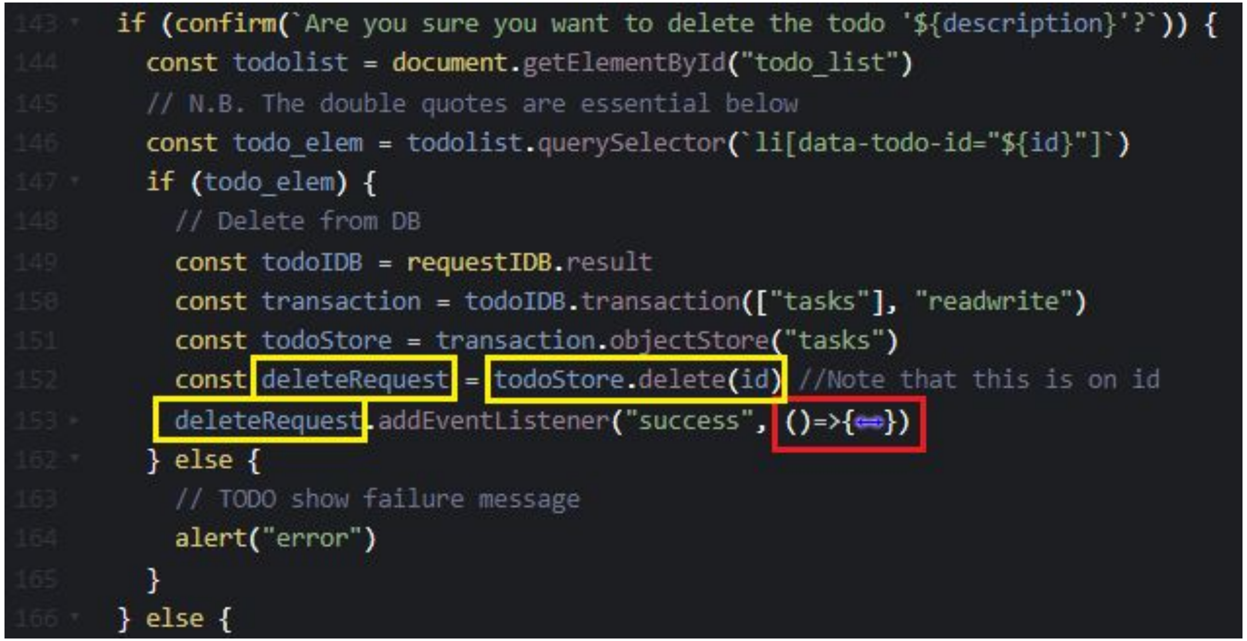
University of Sheffield

Sprint 5 - deleting from indexeddb

Again folded

This is another transaction based operation …

N.B. delete is called on id - not on todo



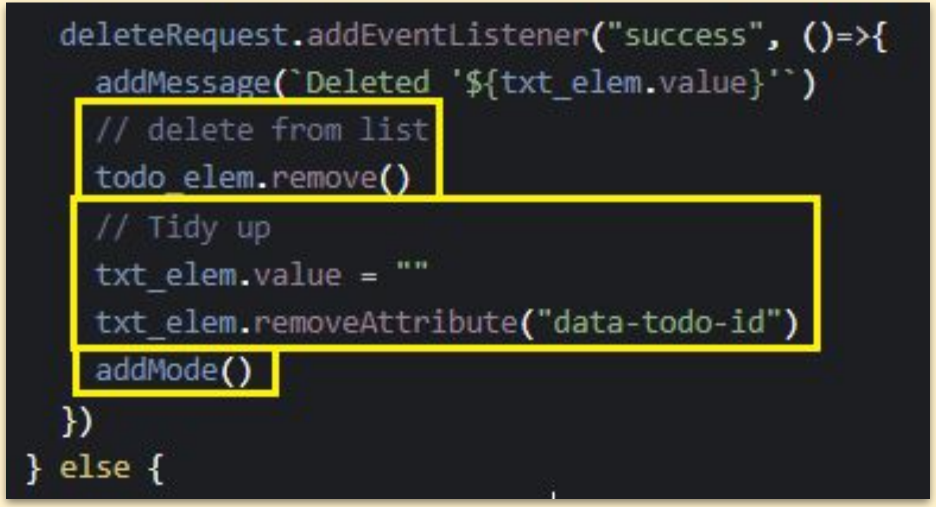
```
if (confirm(`Are you sure you want to delete the todo '${description}'?`)) {
  const todolist = document.getElementById("todo_list")
  // N.B. The double quotes are essential below
  const todo_elem = todolist.querySelector(`li[data-todo-id="${id}"]`)
  if (todo_elem) {
    // Delete from DB
    const todoIDB = requestIDB.result
    const transaction = todoIDB.transaction(["tasks"], "readwrite")
    const todoStore = transaction.objectStore("tasks")
    const deleteRequest = todoStore.delete(id) //Note that this is on id
    deleteRequest.addEventListener("success", ()=>{↔})
  } else {
    // TODO show failure message
    alert("error")
  }
} else {
```

Sprint 5 - deleting from DOM

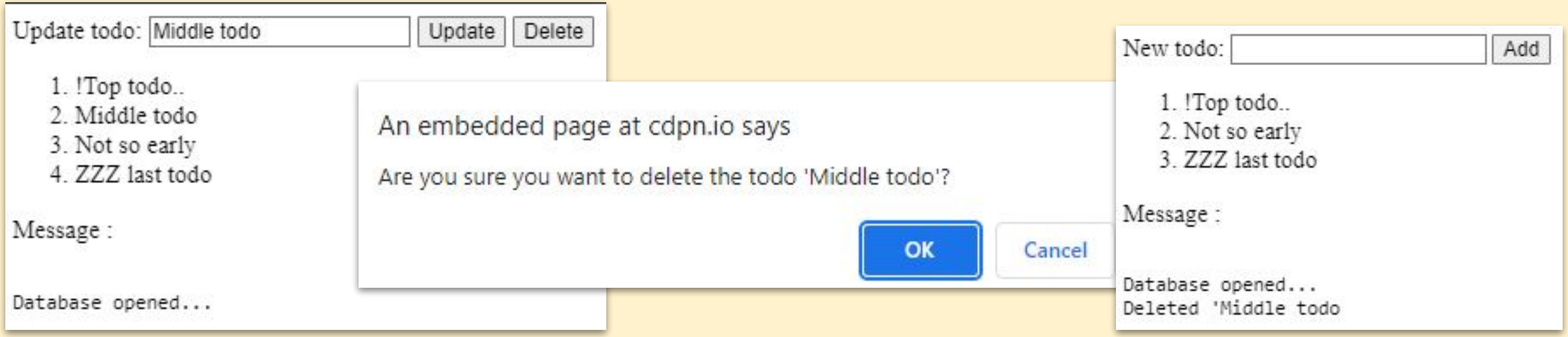
Finally:

Tidy up after a successful indexeddb delete …

Finished - for now :)



```
  deleteRequest.addEventListener("success", ()=>{
    addMessage(`Deleted '${txt_elem.value}'`)
    // delete from list
    todo_elem.remove()
    // Tidy up
    txt_elem.value = ""
    txt_elem.removeAttribute("data-todo-id")
    addMode()
  })
} else {
```



Update todo: Middle todo [Update] [Delete]

1. !Top todo..
2. Middle todo
3. Not so early
4. ZZZ last todo

Message :

```
Database opened...
```

An embedded page at cdpn.io says

Are you sure you want to delete the todo 'Middle todo'?

[OK] [Cancel]

New todo: [Add]

1. !Top todo..
2. Not so early
3. ZZZ last todo

Message :

```
Database opened...
Deleted 'Middle todo
```