Client Side Storage

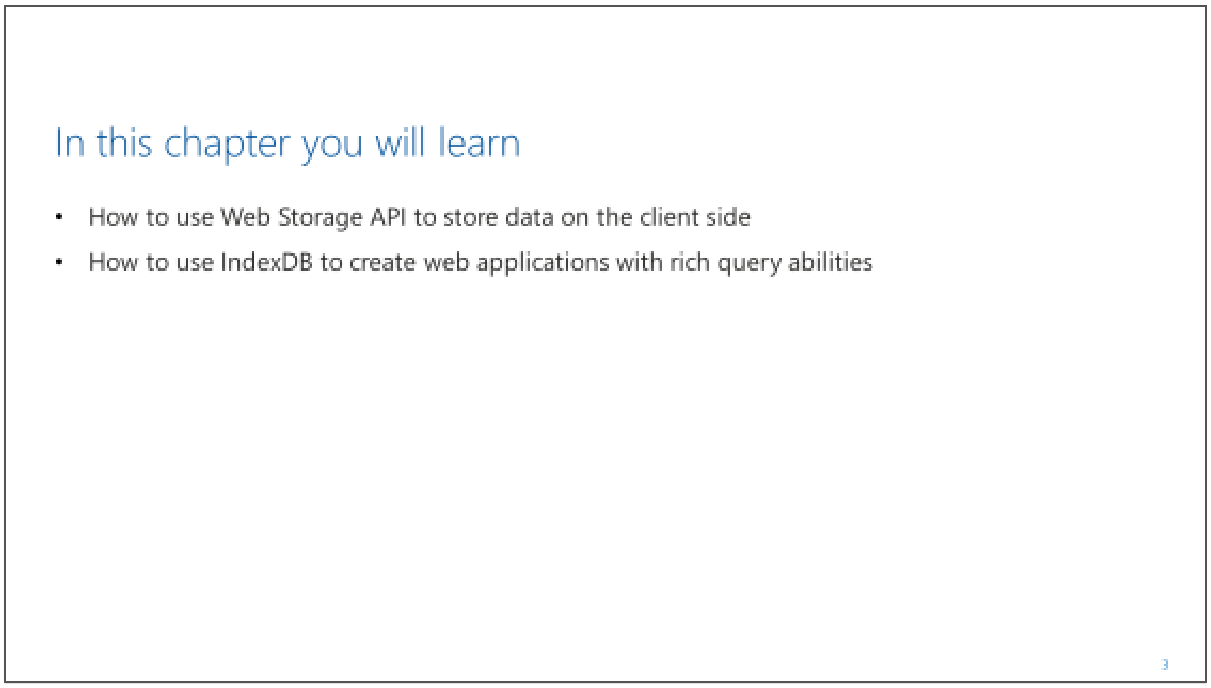
PROGRAMMING WITH JAVASCRIPT

QA

This chapter covers

- Why local storage?
- Web Storage
- IndexedDB

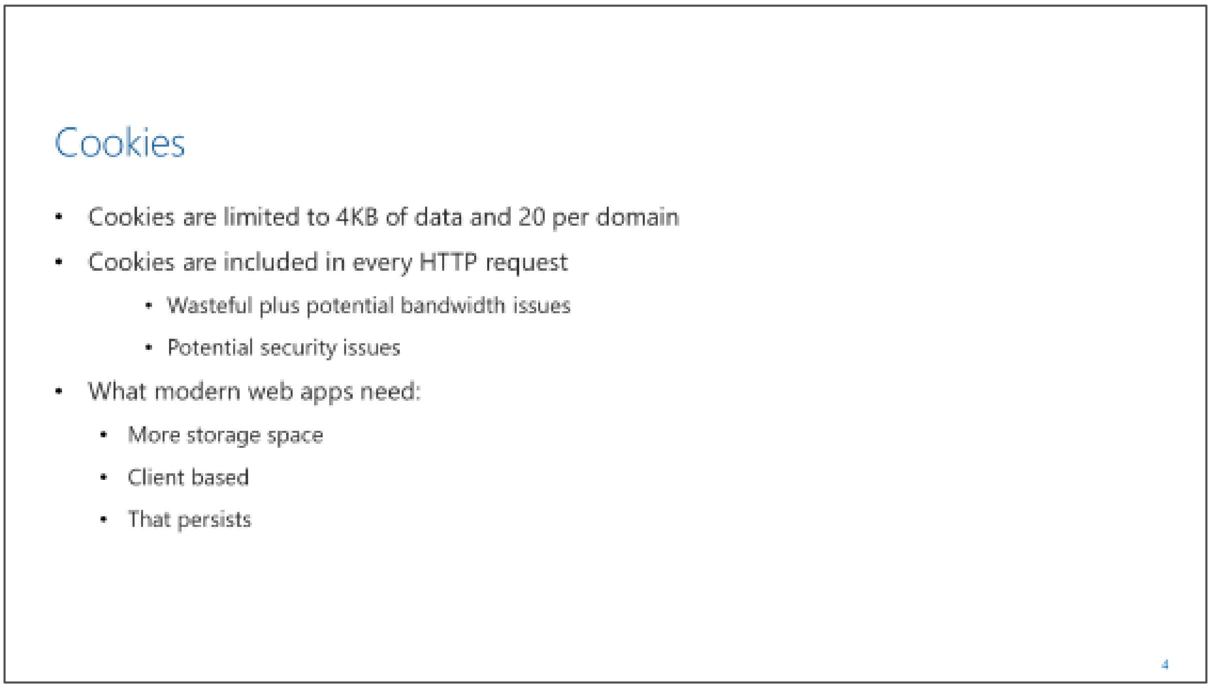
2



In this chapter you will learn

- How to use Web Storage API to store data on the client side
- How to use IndexDB to create web applications with rich query abilities

3



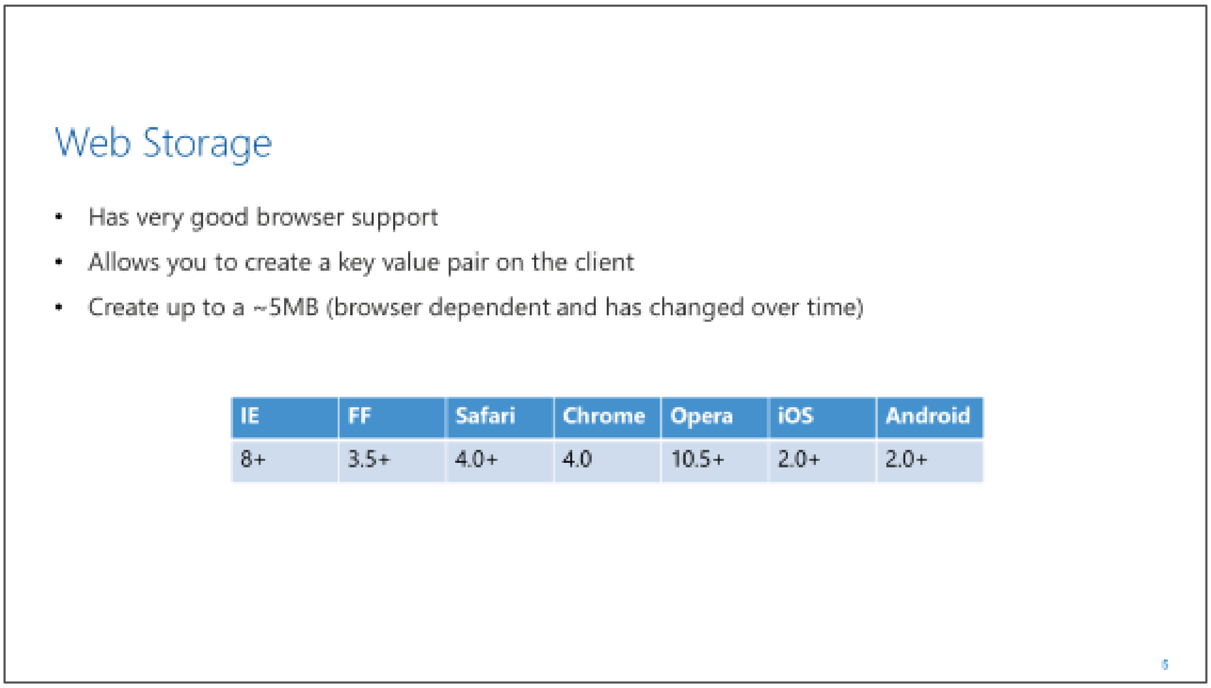
Cookies

- Cookies are limited to 4KB of data and 20 per domain
- Cookies are included in every HTTP request
    - Wasteful plus potential bandwidth issues
    - Potential security issues
- What modern web apps need:
    - More storage space
    - Client based
    - That persists

4

Cookies have been with us for a long time, they are venerable soldiers in the campaign to make a stateless protocol stateful. They have limitations though and some of those limitations can be addressed with other options you may not have heard of.

Be aware that there is no silver bullet here, there are 3 solutions which are suitable in different scenarios – it is up to you, dear reader, to decide which one fits best with your current problem.



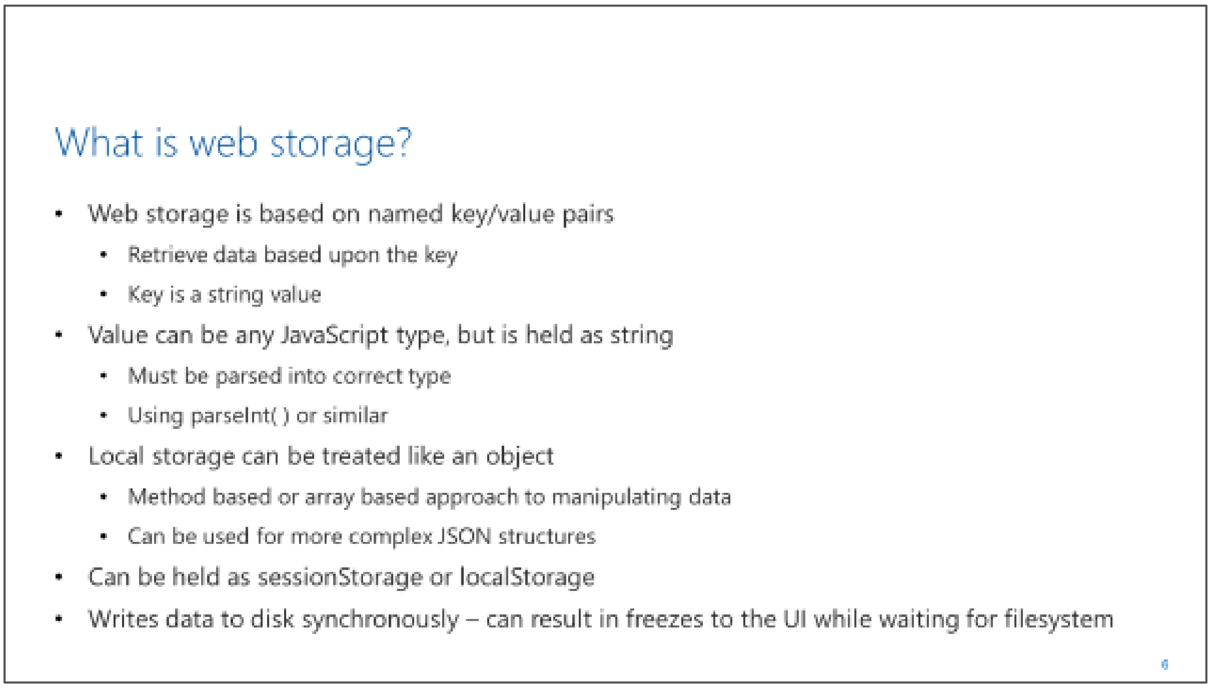
Web Storage

- Has very good browser support
- Allows you to create a key value pair on the client
- Create up to a ~5MB (browser dependent and has changed over time)

IE	FF	Safari	Chrome	Opera	iOS	Android
8+	3.5+	4.0+	4.0	10.5+	2.0+	2.0+

5

The Web Storage API provides mechanisms by which browsers can store key/value pairs, in a much more intuitive fashion than using cookies.



What is web storage?

- Web storage is based on named key/value pairs
  - Retrieve data based upon the key
  - Key is a string value
- Value can be any JavaScript type, but is held as string
  - Must be parsed into correct type
  - Using parseInt( ) or similar
- Local storage can be treated like an object
  - Method based or array based approach to manipulating data
  - Can be used for more complex JSON structures
- Can be held as sessionStorage or localStorage
- Writes data to disk synchronously – can result in freezes to the UI while waiting for filesystem

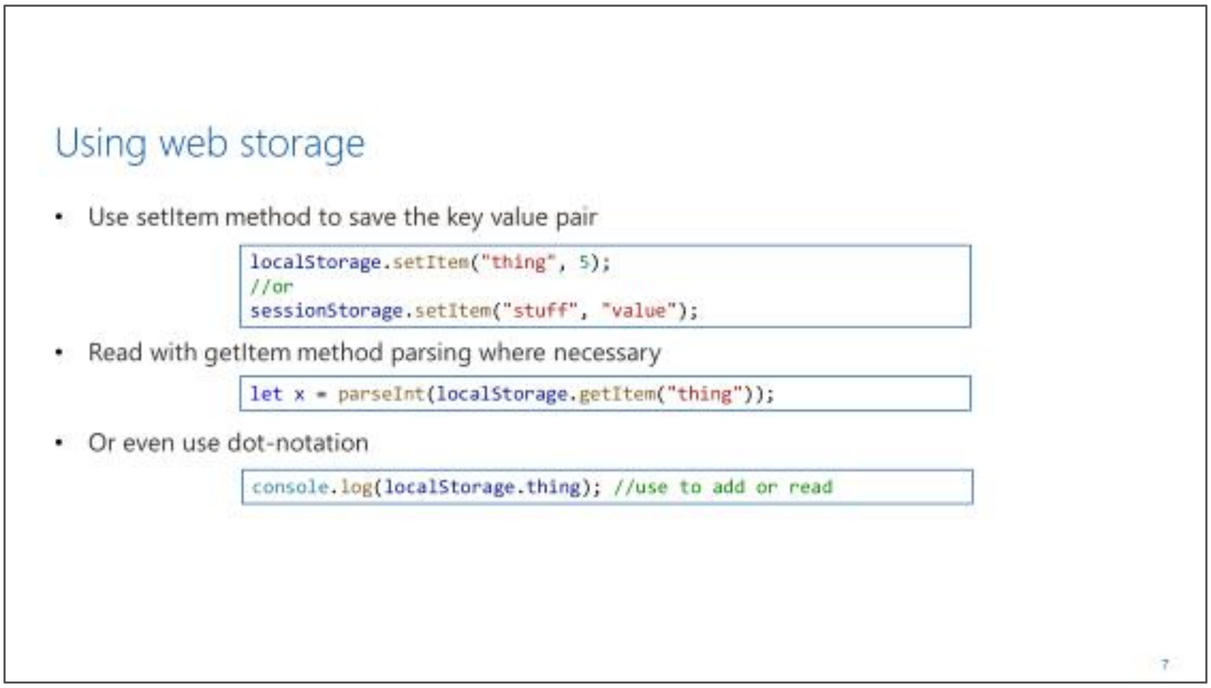
6

Web storage is based on named key/value pairs. Data is stored based on a named key. The key name itself is held as a string but the data can be any type supported by JavaScript, including strings, Booleans, integers, or floats.

The data is actually stored as a string. If you are storing and retrieving anything other than strings, you will need to use functions like parseInt() or parseFloat() to coerce your retrieved data into the expected JavaScript datatype.

The storage API offers two modes sessionStorage and localStorage. Session Storage is available only to the window until the window is closed. Use this for shopping basket type activities, where you would not want the date to leak from one window to another.

localStorage is based around a domain and spans all windows that are open on that domain. The data is cross-session durable and will last until you want to be rid of it.



Using web storage

- Use setItem method to save the key value pair

```
localStorage.setItem("thing", 5);
//or
sessionStorage.setItem("stuff", "value");
```

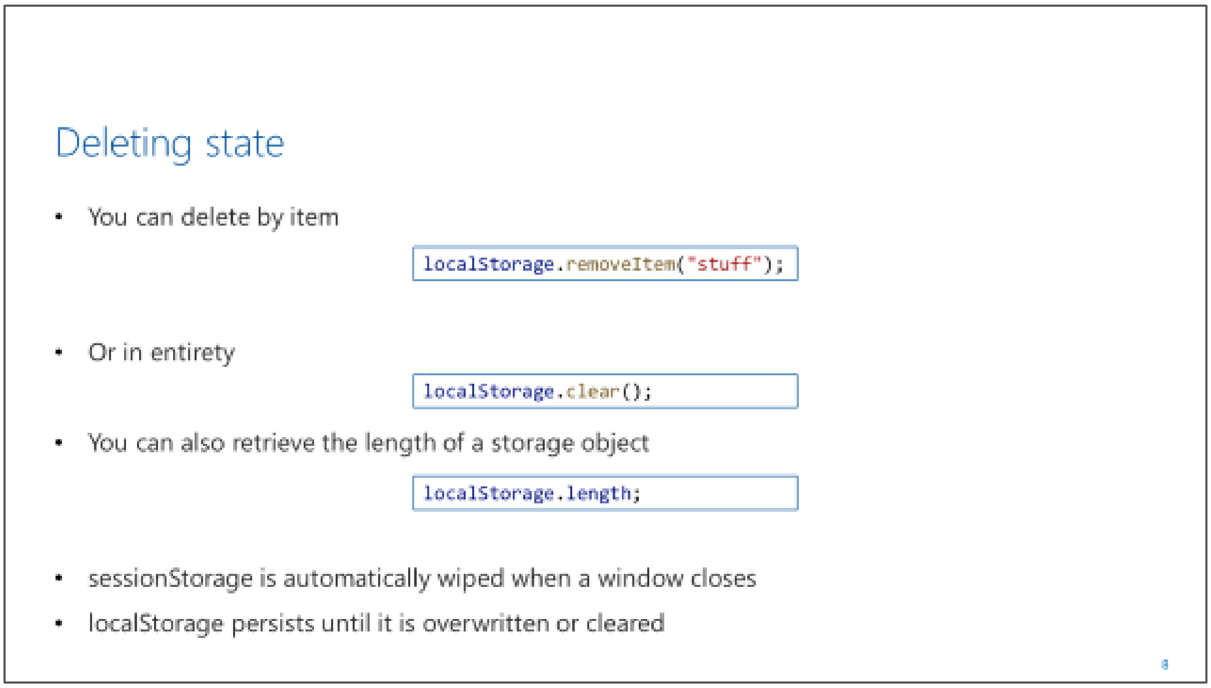
- Read with getItem method parsing where necessary

```
let x = parseInt(localStorage.getItem("thing"));
```

- Or even use dot-notation

```
console.log(localStorage.thing); //use to add or read
```

7



Deleting state

- You can delete by item

```
localStorage.removeItem("stuff");
```

- Or in entirety

```
localStorage.clear();
```

- You can also retrieve the length of a storage object

```
localStorage.length;
```

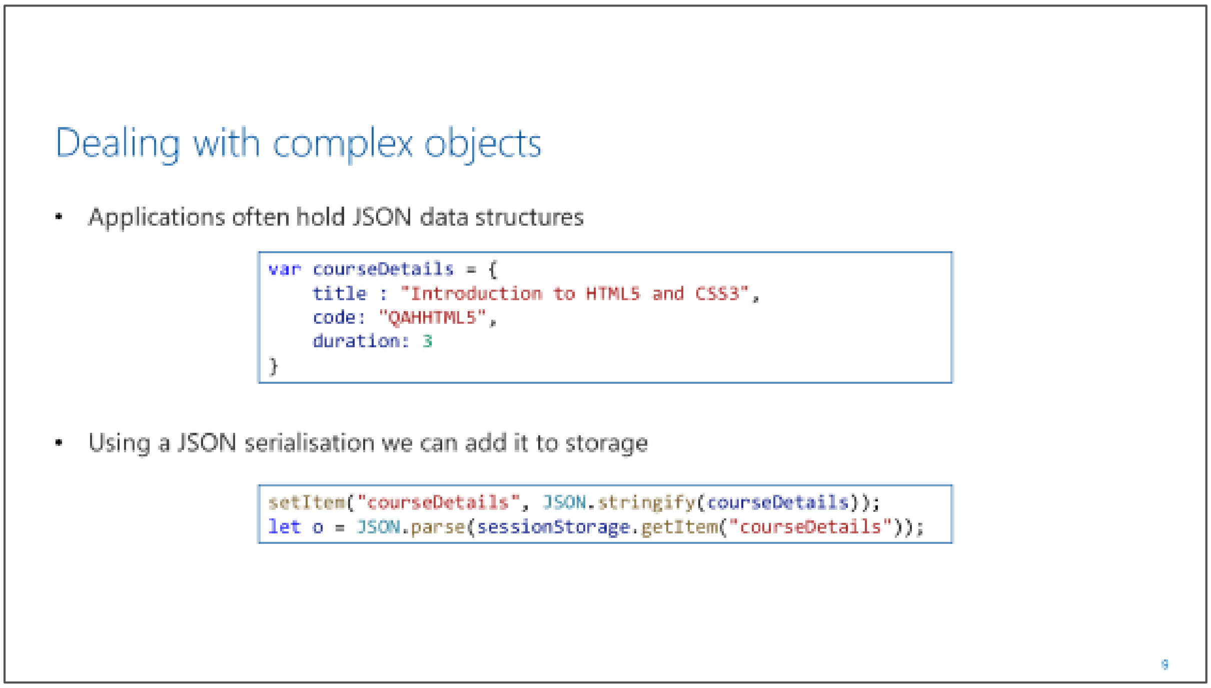
- sessionStorage is automatically wiped when a window closes
- localStorage persists until it is overwritten or cleared

8

Remember local storage persists indefinitely, so you will want to be able to clear it up by key or give a user the choice to wipe out all data.

If the key does not exist, the delete will do nothing.

Be careful in the development lifecycle to ensure you do not use the same key names or objects elsewhere in your application to avoid overwriting values accidently.



Dealing with complex objects

- Applications often hold JSON data structures

```
var courseDetails = {
    title : "Introduction to HTML5 and CSS3",
    code: "QAHHTML5",
    duration: 3
}
```

- Using a JSON serialisation we can add it to storage

```
setItem("courseDetails", JSON.stringify(courseDetails));
let o = JSON.parse(sessionStorage.getItem("courseDetails"));
```

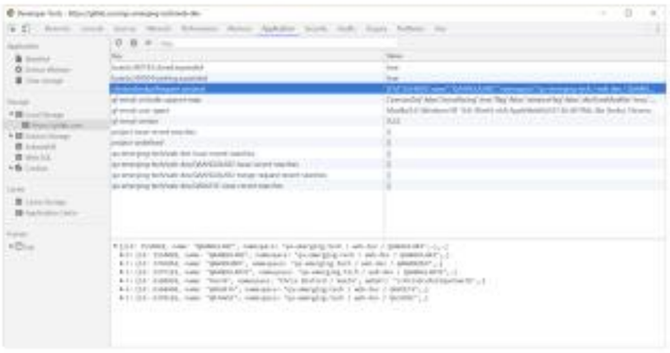
9

In a modern applications, JSON objects are likely being thrown back and forth between client and server on a regular basis. One of the key problems in AJAX application development has been holding state on the client. Web storage makes this a much simpler.

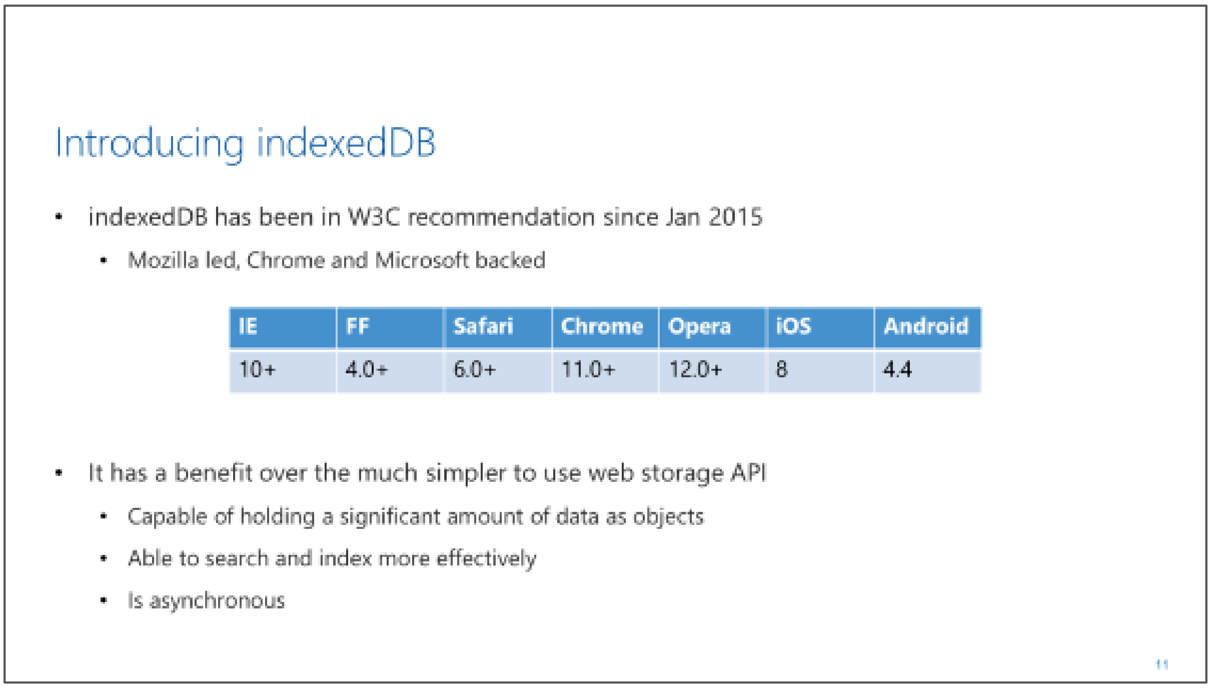
All a developer needs is a way to serialise the JSON object into a string.

Managing state

- You will want to debug and explore web storage state. Many of the browsers have a way of doing this



10



Introducing indexedDB

- indexedDB has been in W3C recommendation since Jan 2015
  - Mozilla led, Chrome and Microsoft backed

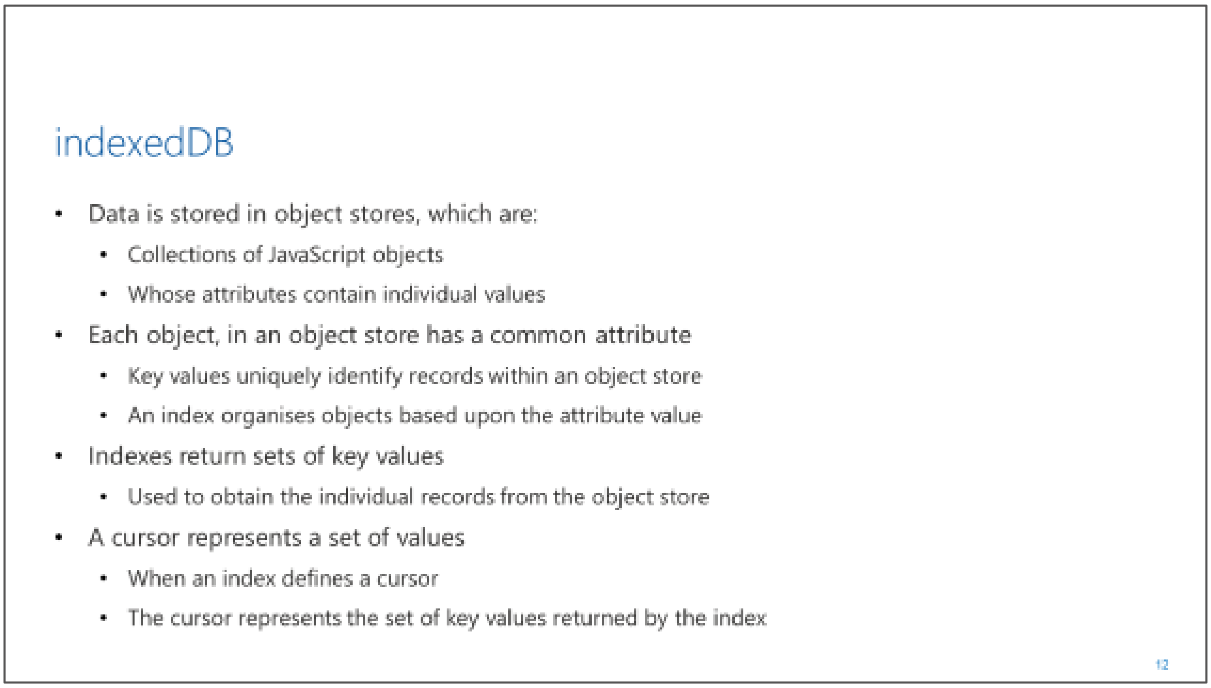
IE	FF	Safari	Chrome	Opera	iOS	Android
10+	4.0+	6.0+	11.0+	12.0+	8	4.4

- It has a benefit over the much simpler to use web storage API
  - Capable of holding a significant amount of data as objects
  - Able to search and index more effectively
  - Is asynchronous

11

The asynchronous API is a non-blocking system and as such will not get data through return values, but rather will get data delivered to a defined callback function.

The IndexedDB support through HTML is transactional. It is not possible to execute commands or open cursors outside of a transaction. There are several types of transactions: read/write transactions, read only and snapshot. In this tutorial, we will be using read/write transactions.



indexedDB

- Data is stored in object stores, which are:
  - Collections of JavaScript objects
  - Whose attributes contain individual values
- Each object, in an object store has a common attribute
  - Key values uniquely identify records within an object store
  - An index organises objects based upon the attribute value
- Indexes return sets of key values
  - Used to obtain the individual records from the object store
- A cursor represents a set of values
  - When an index defines a cursor
  - The cursor represents the set of key values returned by the index

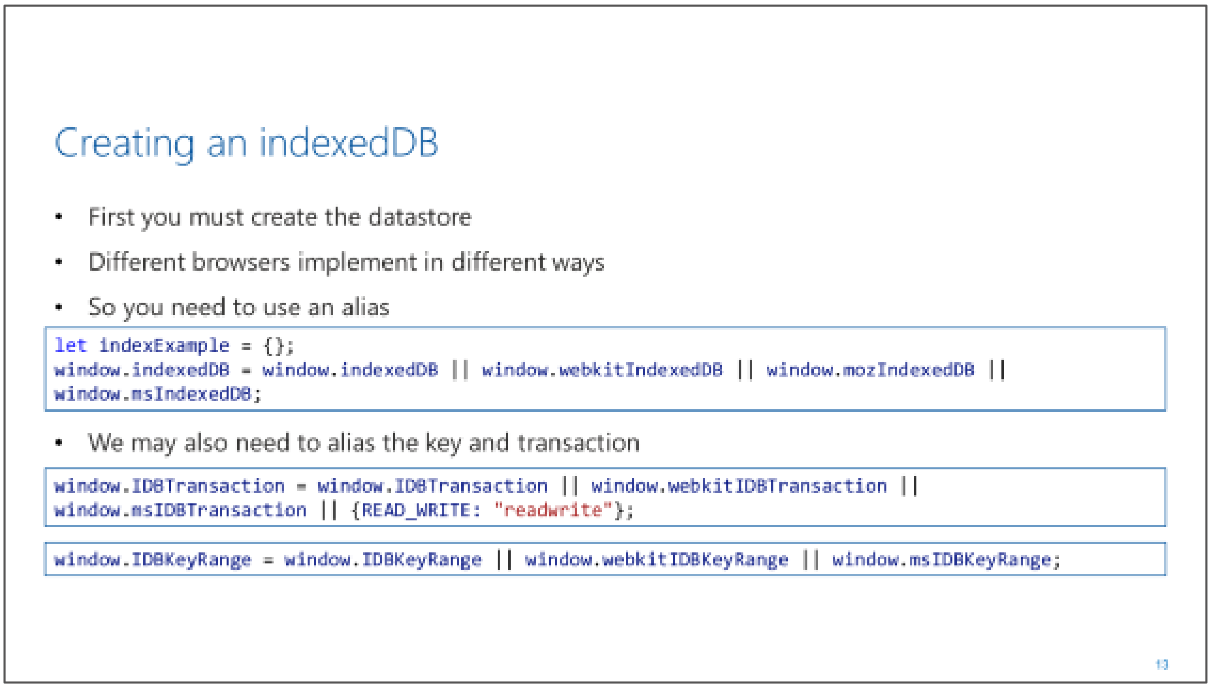
12

IndexedDB is an Object Store. It is not the same as a Relational Database, which has tables, with collections rows and columns. It is an important and fundamental difference and affects the way that you design and build your applications.

In a traditional relational data store, we would have a table of "to do" items that store a collection of the users to do data in rows. With columns of named types of data. To insert data, the semantics normally follow: INSERT INTO Todo(id, data, update_time) VALUES (1, "Test", "01/01/2010");

IndexedDB differs in that you create an Object Store for a type of data and simply persist Javascript Objects to that store. Each Object Store can have a collection of Indexes that make it efficient to query and iterate across.

IndexedDB also does away with the notion of a Standard Query Language (SQL), instead it is replaced with a query against an index which produces a cursor that you use to iterate across the result set.



Creating an indexedDB

- First you must create the datastore
- Different browsers implement in different ways
- So you need to use an alias

```
let indexExample = {};
window.indexedDB = window.indexedDB || window.webkitIndexedDB || window.mozIndexedDB ||
window.msIndexedDB;
```

- We may also need to alias the key and transaction

```
window.IDBTransaction = window.IDBTransaction || window.webkitIDBTransaction ||
window.msIDBTransaction || {READ_WRITE: "readwrite"};
```

```
window.IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange || window.msIDBKeyRange;
```

13

Data is stored in object stores, which are collections of JavaScript objects, whose attributes contain individual values.

Each JavaScript object, sometimes called a record, in an object store has a common attribute called a key path; the value of this attribute is called a key value (or key). Key values uniquely identify individual records within an object store.

An index organises objects according to the value of a common attribute. Indexes return sets of key values that can be used to obtain the individual records from the original object store.

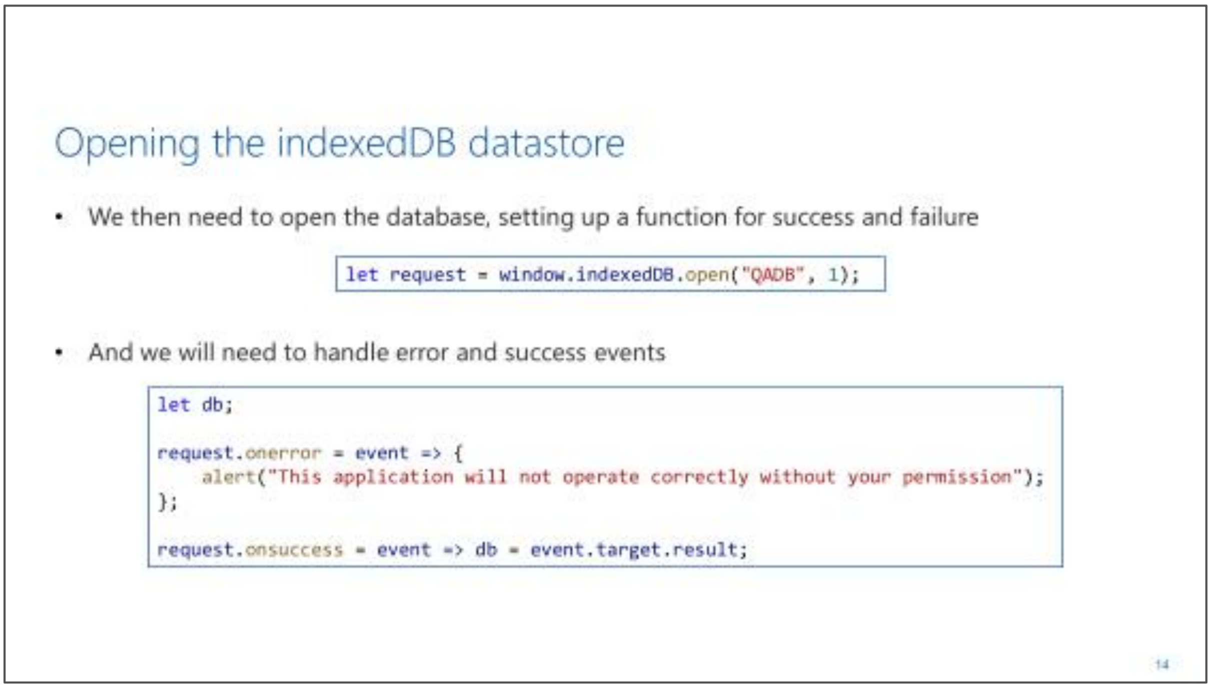
A cursor represents a set of values. When an index defines a cursor, the cursor represents the set of key values returned by the index. When an object store defines a cursor, the cursor represents a set of records stored in the cursor.

A keyRange defines a range of values for an index or a set of records in an object store; key ranges allow you to filter cursor results.

A database contains the object stores and indexes; databases also manage transactions.

A request represents individual actions taken against objects in a database. For example, opening a database leads to a request object and you define event handlers on the request object to react to the results of the request.

A transaction manages the context of operations and is used to maintain the integrity of database activities. For example, you can create object stores only in the context of a version change transaction. If a transaction is aborted, all operations within the transaction are cancelled.



Opening the indexedDB datastore

- We then need to open the database, setting up a function for success and failure

```
let request = window.indexedDB.open("QADB", 1);
```

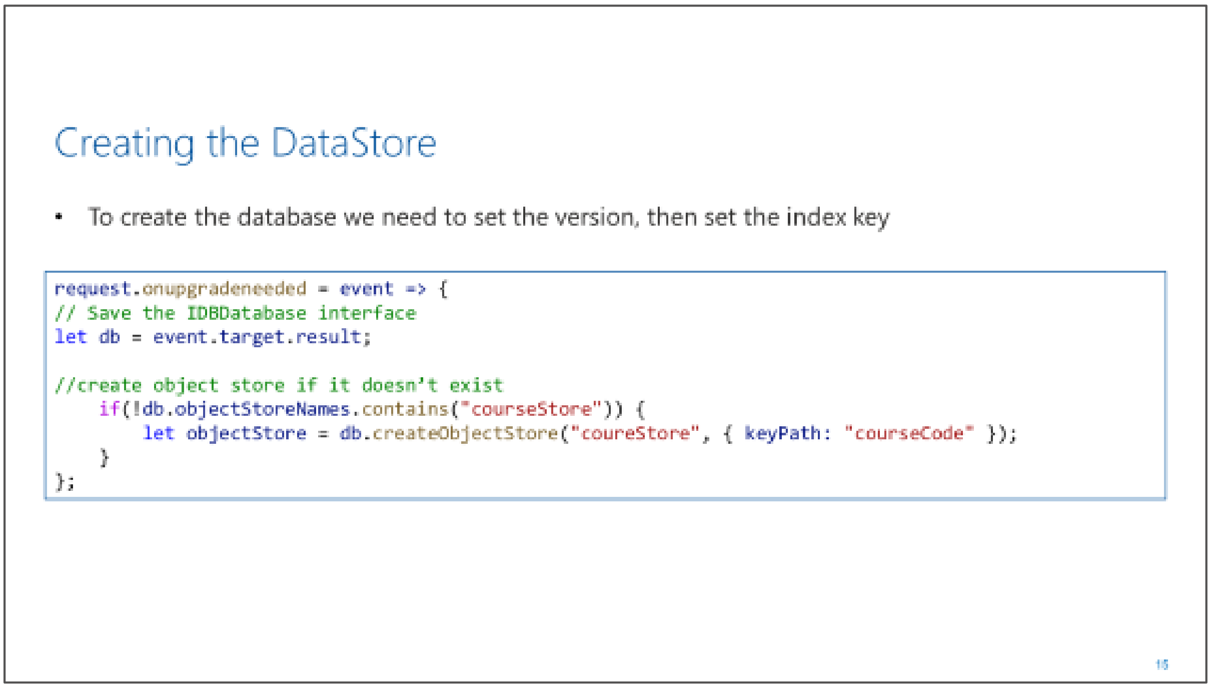
- And we will need to handle error and success events

```
let db;

request.onerror = event => {
    alert("This application will not operate correctly without your permission");
};

request.onsuccess = event => db = event.target.result;
```

14

In the above example, the asynchronous pattern for indexedDB is being used. First you assign a function to the open event of the indexedDB. If the request to open the database is successful, the onsuccess event is fired where the success passes a reference to the database we wish to work with.

If any error is received, we trap it with the onfaliure event assigning it to the on error event of the indexedDB object.



Creating the DataStore

- To create the database we need to set the version, then set the index key

```
request.onupgradeneeded = event => {
// Save the IDBDatabase interface
let db = event.target.result;

//create object store if it doesn't exist
    if(!db.objectStoreNames.contains("courseStore")) {
        let objectStore = db.createObjectStore("coureStore", { keyPath: "courseCode" });
    }
};
```

15

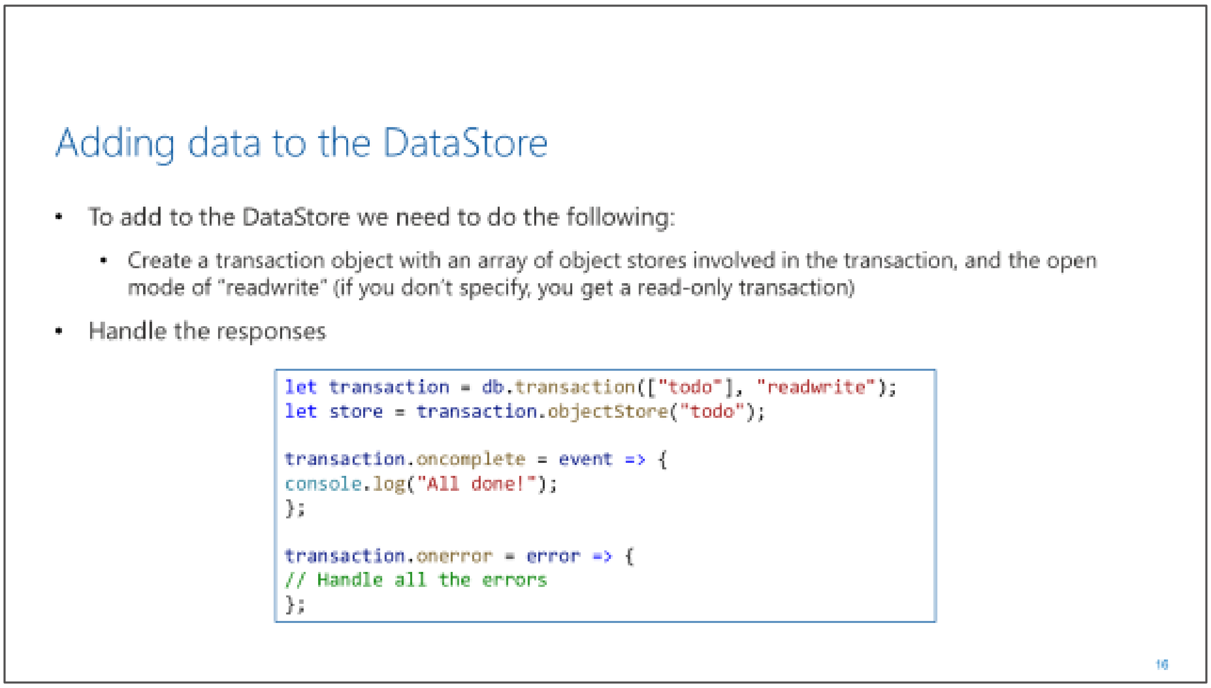
The above code actually does quite a lot. We define an "open" method in our API, which when called will open the database "dbName". The open request isn't executed straight away. Instead, an IDBRequest is returned.

The indexedDB.open method will be called when the current function exits. This is a little different to how we normally assign asynchronous callbacks, but we get the chance to attach our own listeners to the IDBRequest object before the callbacks are executed.

If the open request is successful, our onsuccess callback is executed. In this callback we check the database version and if it is not the same as the number we expect, we call "setVersion".

SetVersion is the only place in our code that we can alter the structure of the database. In it we can create and delete Object Stores and build and remove indexes. A call to setVersion returns an IDBRequest object where we can attach our callbacks. When successful, we start to create our Object Stores.

Object Stores are created with a single call to createObjectStore. The method takes a name of the store, and a parameter object. The parameter object is very important, it lets you define important optional properties. In our case, we define a keyPath that is the property that makes an individual object in the store unique. That property in this example is "timeStamp". "timeStamp" must be present on every object that is stored in the objectStore.



Adding data to the DataStore

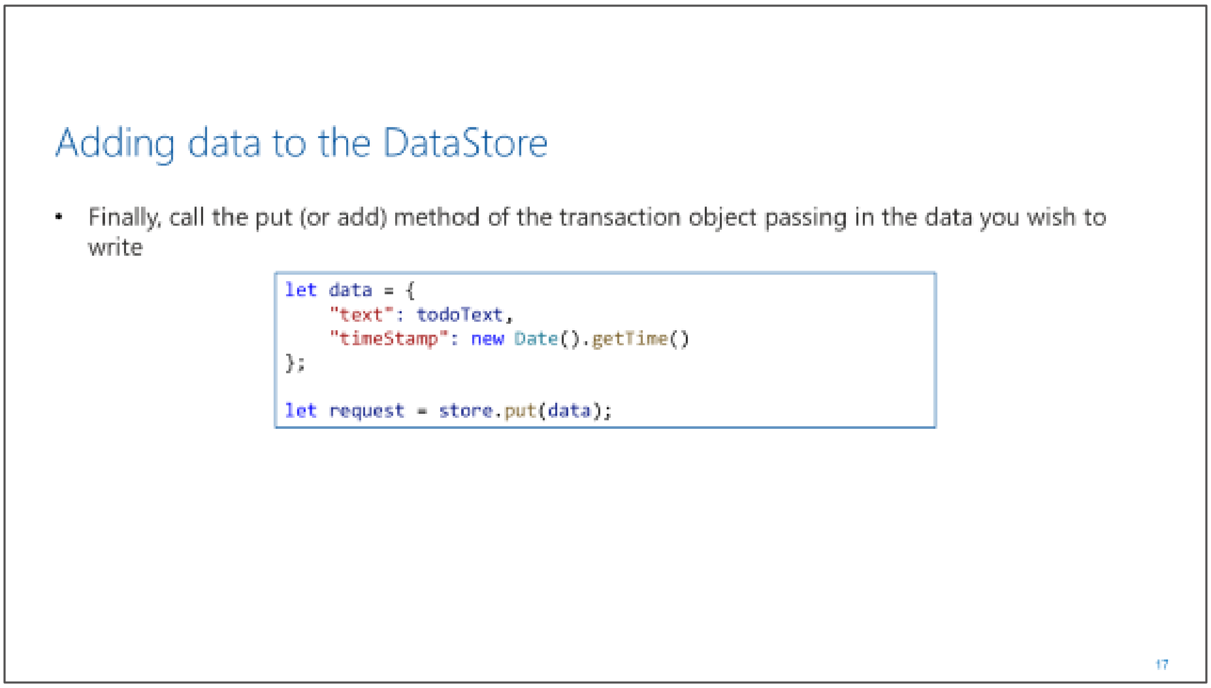
- To add to the DataStore we need to do the following:
  - Create a transaction object with an array of object stores involved in the transaction, and the open mode of "readwrite" (if you don't specify, you get a read-only transaction)
- Handle the responses

```
let transaction = db.transaction(["todo"], "readwrite");
let store = transaction.objectStore("todo");

transaction.oncomplete = event => {
console.log("All done!");
};

transaction.onerror = error => {
// Handle all the errors
};
```

16

Now that the application has access to the Object Store, we can issue a simple put command with a basic JSON object.

```
request.onsuccess = function(e) {
         //code to update ui
};
request.onerror = function(e) {
         //error handeling code
};
```



Adding data to the DataStore

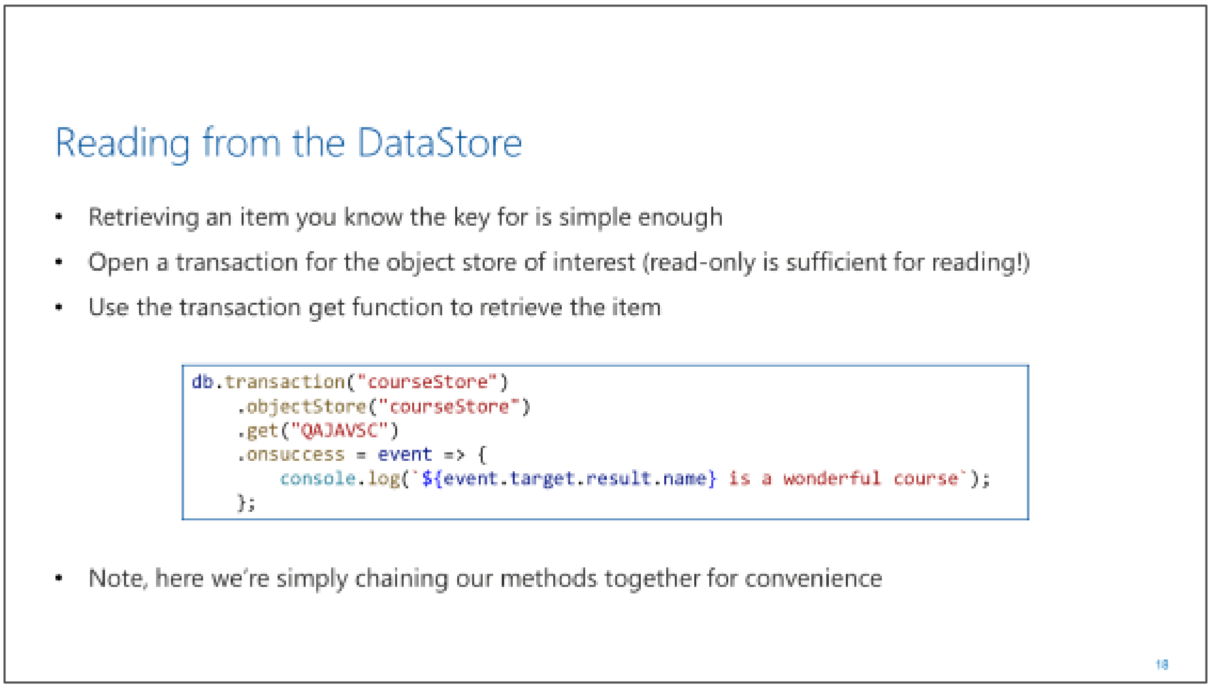
- Finally, call the put (or add) method of the transaction object passing in the data you wish to write

```
let data = {
    "text": todoText,
    "timeStamp": new Date().getTime()
};

let request = store.put(data);
```

17

Now that the application has access to the Object Store, we can issue a simple put command with a basic JSON object.

```
request.onsuccess = function(e) {
         //code to update ui
};
request.onerror = function(e) {
         //error handeling code
};
```



Reading from the DataStore

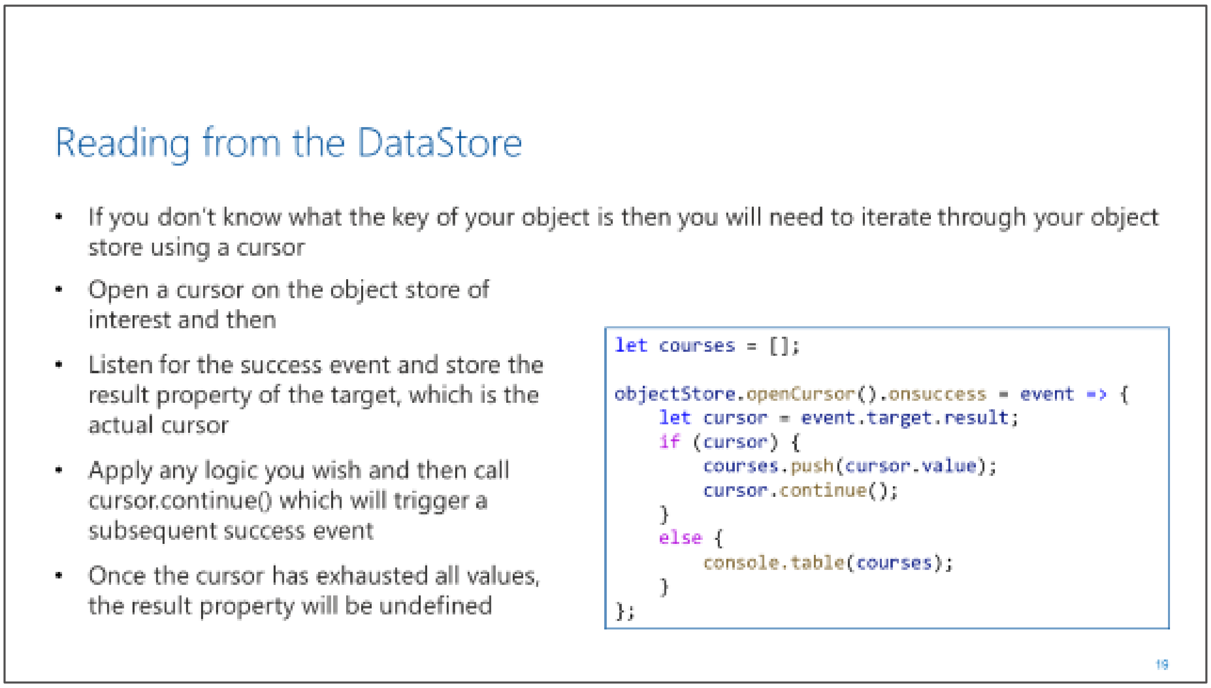
- Retrieving an item you know the key for is simple enough
- Open a transaction for the object store of interest (read-only is sufficient for reading!)
- Use the transaction get function to retrieve the item

```
db.transaction("courseStore")
    .objectStore("courseStore")
    .get("QAJAVSC")
    .onsuccess = event => {
        console.log(`${event.target.result.name} is a wonderful course`);
    };
```

- Note, here we're simply chaining our methods together for convenience

18

Constant	Value	Description
READ_ONLY	0	Allows data to be read but not changed.
READ_WRITE	1	Allows reading and writing of data in existing data stores to be changed.
VERSION_CHANGE	2	Allows any operation to be performed, including ones that delete and create object stores and indexes.



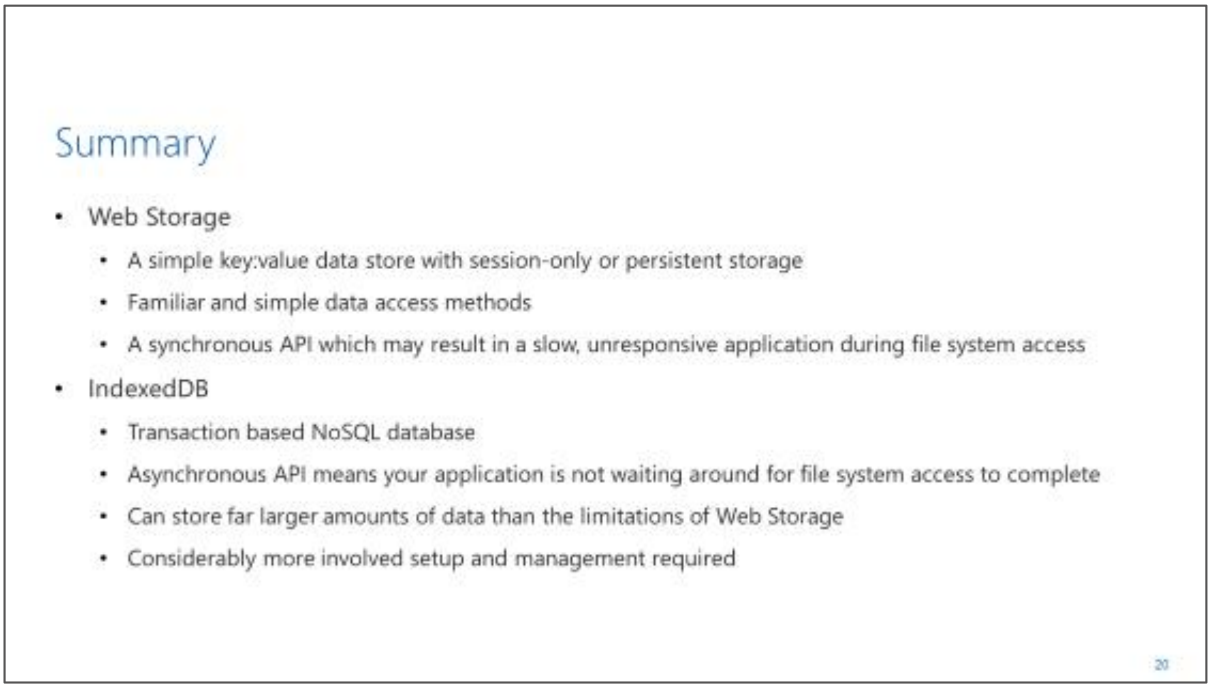
Reading from the DataStore

- If you don't know what the key of your object is then you will need to iterate through your object store using a cursor
- Open a cursor on the object store of interest and then
- Listen for the success event and store the result property of the target, which is the actual cursor
- Apply any logic you wish and then call cursor.continue() which will trigger a subsequent success event
- Once the cursor has exhausted all values, the result property will be undefined

```
let courses = [];

objectStore.openCursor().onsuccess = event => {
    let cursor = event.target.result;
    if (cursor) {
        courses.push(cursor.value);
        cursor.continue();
    }
    else {
        console.table(courses);
    }
};
```

19

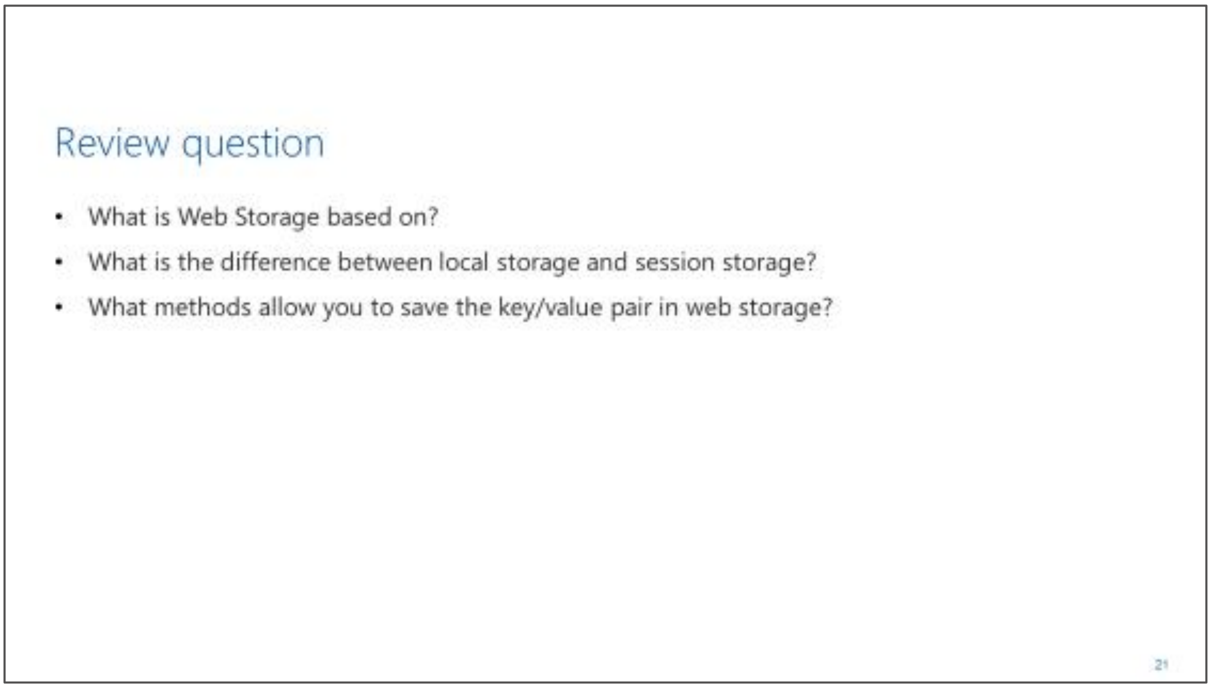
Constant	Value	Description
READ_ONLY	0	Allows data to be read but not changed.
READ_WRITE	1	Allows reading and writing of data in existing data stores to be changed.
VERSION_CHANGE	2	Allows any operation to be performed, including ones that delete and create object stores and indexes.



Summary

- Web Storage
  - A simple key:value data store with session-only or persistent storage
  - Familiar and simple data access methods
  - A synchronous API which may result in a slow, unresponsive application during file system access
- IndexedDB
  - Transaction based NoSQL database
  - Asynchronous API means your application is not waiting around for file system access to complete
  - Can store far larger amounts of data than the limitations of Web Storage
  - Considerably more involved setup and management required

20



Review question

- What is Web Storage based on?
- What is the difference between local storage and session storage?
- What methods allow you to save the key/value pair in web storage?

21