1. **Ajax with XHR**

<https://www.w3schools.com/xml/ajax_xmlhttprequest_create.asp>

http.onreadystatechange = function() {

if (this.readyState == 4 && this.status == 200) {

document.getElementById("demo").innerHTML = this.responseText;

}

};

xhttp.open("GET", "ajax\_info.txt", true);

xhttp.send();

1. **AJAX with JQuery**

The [**.ajax() method**](http://api.jquery.com/jquery.ajax/) is at the heart of all asynchronous requests for the entire jQuery library. There are a couple of ways you can call the .ajax() method:

$.ajax(<url-to-fetch>, <a-configuration-object>);

*// or*

$.ajax(<just a configuration object>);

The most common way to use the .ajax() method is with just the configuration object, since everything can be set inside the configuration object.

**What's a "configuration object"?**

A configuration object is just a plain ol' JavaScript object that's used to configure something. For example:

**var** settings = {

frosting: 'buttercream',

colors: ['orange', 'blue'],

layers: 2,

isRound: true

};

...the settings configuration object can be used in the imaginary MakeCake constructor function:

**const** myDeliciousCake = MakeCake( settings );

Alternatively, the settings object could be passed in directly:

**const** myDeliciousCake = MakeCake({

frosting: 'buttercream',

colors: ['orange', 'blue'],

layers: 2,

isRound: true

});

## Makig an Ajax call

1. jQuery's .ajax() method has to be incredibly versatile and powerful if it's what powers all of jQuery's asynchronous requests. A simple Ajax request would look like this:
2. $.ajax({
3. url: 'https://swapi.co/api/people/1/'
4. });

**3. AJAX fetch**

[**https://developer.mozilla.org/en-US/docs/Web/API/Fetch\_API**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

Fetch is **promise based**. Might need **Polyfill.**

fetch('https://api.unsplash.com/search/photos?page=1&query=flowers');

**Fetch requests still need to obey the cross-origin protocol** of how resources are shared. This means that, by default, you can only make requests for assets and data on the same domain as the site that will end up loading the data.

Deafult method is **GET** you can change it with the following.

fetch(`https://api.unsplash.com/search/photos?page=1&query=${searchedForText}`, {

method: 'POST'

});

You can change the **headers**  as well in the request

fetch(https://api.unsplash.com/search/photos?page=1&query=${searchedForText}, {   
     headers: {   
         Authorization: 'Client-ID abc123'   
    }   
});

const requestHeaders = new Headers();   
requestHeaders.append('Authorization', 'Client-ID abc123');   
fetch(https://api.unsplash.com/search/photos?page=1&query=${searchedForText}, {   
     headers: requestHeaders   
});

**Handle the received promise from the fetch request**

fetch(`https://api.unsplash.com/search/photos?page=1&query=${searchedForText}`, {

headers: {

Authorization: 'Client-ID abc123'

}

}).then(**function**(response) {

**debugger**; *// work with the returned response*

});

**Handling the received data in the response**

fetch(`https://api.unsplash.com/search/photos?page=1&query=${searchedForText}`, {

headers: {

Authorization: 'Client-ID abc123'

}

}).then(**function**(response) {

**return** response.json();

}).then(addImage);

**function** **addImage**(data) {

**debugger**;

}

1. So we initiate a **fetch request**
2. It will return a promise, with the **response**
3. We have to use the response’s **.josn()**  method to get the body of the response
4. Then we can execute the added **callback**

**Handling error with catch()**

fetch(`https://api.unsplash.com/search/photos?page=1&query=${searchedForText}`, {

headers: {

Authorization: 'Client-ID abc123'

}

}).then(response => response.json())

.then(addImage)

.catch(e => requestError(e, 'image'));

**function** **addImage**(data) {

**debugger**;

}

**function** **requestError**(e, part) {

console.log(e);

responseContainer.insertAdjacentHTML('beforeend', `<p class="network-warning">Oh no! There was an error making a request for the ${part}.</p>`);

}

# **Lesson 5 IndexedDB**

Where you can search for the codes **(MobileWebSpecialist\1\_Why\_Responsive\Lesson\_16\_ServiceWorker)**

Go to thelast commit

git checkout task-cache-avatars

1. **Creating a new key-value object store**
2. **Read "hello" in "keyval" indexed db**
3. **Add a keyval to the objectstore with put**
4. **Building up an indexedDB version control,**
5. Add objects to the objectstore
6. Selecting an indexed object store and getting elements by their key indexes.
7. Looping through each item of the indexed DB with openCursor()
8. Interacting with service workers
9. cache photos

Every code is executed at the folllowing file

**(((((public/js/idb-test.js))))**

------**//// Creating a indexedDB + adding some keyvalobject store values /////**\*------

\* Import idb module, from node\_modules

import idb from 'idb';

**Notes:**

* Transactions are atomic, so it only happens and give back completed promise if all of the transactions took place correctly
* New ObjectStores can be created only when the upgradeDB is called.

1. **Creating a new key-value object store**

// idb.open returns a promise, which is a database object, which we store for future use.

// Inital a datapase, (name,versions,callback) to setup the database

, version start with 1

// The only place where you can create, or remove ojbect stores, databases

is the upgradeDB funciton.

// we create a key-value store and put an object of it.

var dbPromise = idb.open('test-db', 1, function(upgradeDb) {

var keyValStore = upgradeDb.createObjectStore('keyval');

keyValStore.put("world", "hello");

});

1. **Read "hello" in "keyval" indexed db**

// So we are creating using the returned promise, then we execute an anonymus funciton

dbPromise.then(function(db) {

var tx = db.transaction('keyval');

// First create a transaction, then determine which objectstor to use,

var keyValStore = tx.objectStore('keyval');

//then use the objectstore's function to get the information

return keyValStore.get('hello');

//Get the hello key

/ everything will be returned as a promise, so by using again the then, we can get the returned value form the promise

}).then(function(val) {

console.log('The value of "hello" is:', val);

});

1. **Add a keyval to the objectstore with put + concatenating the function not to declare multiple variable**

dbPromise.then(function(db) {

var tx = db.transaction('keyval', 'readwrite');

//Create a transaction

var keyValStore = tx.objectStore('keyval');

// Select ojbect store

keyValStore.put('bar', 'foo'); //Put Value into it

return tx.complete; //Return if it is completed

}).then(function() {

console.log('Added foo:bar to keyval');

});

//concatenating the function not to declare anything twice

dbPromise.then(function(db) {

var tx = db.transaction('keyval', 'readwrite').objectStore('keyval').put('cat', 'favoriteAnimal');

return tx.complete;

}).then(function() {

console.log('Added favoriteAnimal:cat to keyval');

});

**4. Building up an indexedDB version control,** with switch case. Thus if we want to add a new objectstore, we can add it and keep all of the preivous code as well.

var dbPromise = idb.open('test-db', 4, function(upgradeDb) {

switch(upgradeDb.oldVersion) {

case 0://Create a simple key-val objectsotre

var keyValStore = upgradeDb.createObjectStore('keyval');

keyValStore.put("world", "hello");

case 1:

//Create an ojbectsotre, where we can assign js ojbects and the **'name'** attribute will be the key to the assigned object

upgradeDb.createObjectStore('people', { keyPath: 'name' });

case 2:

//Create a transaction and create an special indexed order with the designed key

var peopleStore = upgradeDb.transaction.objectStore('people');

peopleStore.createIndex('animal', 'favoriteAnimal');

case 3:

var peopleStore1 = upgradeDb.transaction.objectStore('people');

peopleStore1.createIndex('age','age');

}

});

* 1. **. Add objects to the objectstore**

dbPromise.then(function(db) {

var tx = db.transaction('people', 'readwrite');

var peopleStore = tx.objectStore('people');

peopleStore.put({

name: 'Sam Munoz',

age: 25,

favoriteAnimal: 'dog'

});

peopleStore.put({

name: 'Susan Keller',

age: 34,

favoriteAnimal: 'cat'

});

return tx.complete;

}).then(function() {

console.log('People added');

});

**7.Selecting an indexed object store and getting elements by their key indexes.**

dbPromise.then(function(db) {

// list all cat people

var tx = db.transaction('people').objectStore('people').index('animal');

return animalIndex.getAll('cat');

}).then(function(people) {

console.log('Cat people:', people);

});

dbPromise.then(function(db) {

// TODO: console.log all people ordered by age

var tx = db.transaction('people').objectStore('people').index('age');

return ageIndex.getAll();

}).then(function(people) {

console.log('Age people:', people);

});

***8. Looping through each item of the indexed DB with* openCursor() *+* inside loop with cursor.contiue() + lot of promises**

**-** dbPromise is actually a promise, and we are executing all of the below code upon successfull return

- openCursors() returns again a promise, which will contain the item in the indexdb

- cursor.advance() will also return a promise which will contain desired item in the indexdb

- cursor.continue() will also return a promise which will next desired item in the indexdb

dbPromise.then(function(db) {

var tx = db.transaction('people').objectStore('people').index('age');

//1.Select the indexed objects from a specific object store

return ageIndex.openCursor();

}).then(function(cursor) {

//Return the openCursors promise function

if (!cursor) return; //Undefined cursor means no item is in the function

return cursor.advance(2); //Tell the cursor to skip the first to record

}).then(function logPerson(cursor) {

if (!cursor) return;

console.log("Cursored at:", cursor.value.name);

//Cursor argument in the callback function is the actual item in the db

return cursor.continue().then(logPerson);

//cursor.continue (will go over the next item) on success it will return and in it's callback you have the logperson (inside looop) which will contiue until the cursor.continue will be undefined(no more item)

}).then(function() {

console.log('Done cursoring');

//Then if every promise is eneded, returne, we can consolo log that it has been finished.

});

**Cursor Notes:**

* cursor.update(newValue) --> To change the value, or
* cursor.delete() --> To delete this entry
* cursor.advance(2) --> Skip the first 2 elements
* cursor.continue() --> Will jump to the next item

**9. Interacting with service workers**

1 , Create a datastore, with the idb, wittrs, with an id keypatch, indexed by by-date

this.\_dbPromise = openDatabase(); //Which returns a promise, either successful/failure storage creation or non existing serviceworker support

function openDatabase() {

if (!navigator.serviceWorker) {

//Browser support

return Promise.resolve();

}

return idb.open('wittr', 1, function(upgradeDb) {

var store = upgradeDb.createObjectStore('wittrs', { keyPath: 'id' });

store.createIndex('by-date', 'time');

});

}

2 Then with the recieved websocket json data, you can loop through each item and add them to the list, using transaciton, then put

// called when the web socket sends message data

IndexController.prototype.\_onSocketMessage = function(data) {

var messages = JSON.parse(data);

this.\_dbPromise.then(function(db) {

if (!db) return;

var tx = db.transaction('wittrs', 'readwrite').objectStore('wittrs');

for(var i=0; i<messages.length;i++){

wittrsStore.put(messages[i]);

}

});

this.\_postsView.addPosts(messages);

};

**/////Lesson4.7 Add entries from the database to the cache ///// -**

Simply get all, then and pass it to the prewritten functionc

// TODO: get all of the wittr message objects from indexeddb,

// then pass them to:

// indexController.\_postsView.addPosts(messages)

// in order of date, starting with the latest.

// Remember to return a promise that does all this,

// so the websocket isn't opened until you're done!

var tx = db.transaction('wittrs').objectStore('wittrs').index('by-date');

return tx.getAll().then((messages)=>{

indexController.\_postsView.addPosts(messages.reverse());

});

**10. cache photos**

1. **Create a “fetch event which will listen if we want to fetch an image” intercepting fetch events for the images**

self.addEventListener('fetch', function(event) {

var requestUrl = new URL(event.request.url);

if (requestUrl.origin === location.origin) {

if (requestUrl.pathname === '/') {

event.respondWith(caches.match('/skeleton'));

return;

}

if (requestUrl.pathname.**startsWith('/photos/'))** {

event.respondWith(servePhoto(event.request));

return;

}

}

1. **Save the photos in the cache or fetch from the cache**

var contentImgsCache = 'wittr-content-imgs';

function servePhoto(request) {

var storageUrl = request.url.replace(/-\d+px\.jpg$/, '');

//will open the cache if possible,

return caches.open(contentImgsCache).then(function(cache) {

//will try to match the url and return with the response if possible

return cache.match(storageUrl).then(function(response) {

if (response) return response;

//else it will make the request

return fetch(request).then(function(networkResponse) {

//then will save the photo

cache.put(storageUrl, networkResponse.clone());

return networkResponse;

});

});

});

}