Week 4.a
Persistence in Javascript via IndexedDb

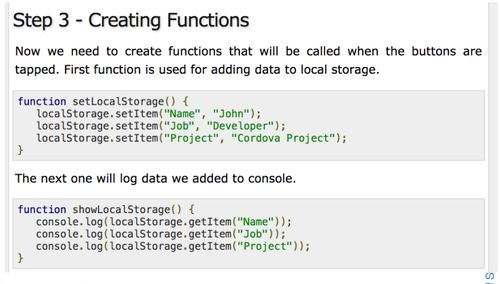
Prof. Fabio Ciravegna
Department of Computer Science
The University of Sheffield
f.ciravegna@shef.ac.uk

Uses

- Personalising site preferences (e.g. showing a user's choice of custom widgets, color scheme, or font size)
- Persisting previous site activity (e.g. storing the contents of a shopping cart from a previous session, remembering if a user was previously logged in).
- Saving data and assets locally so a site will be quicker (and potentially less expensive) to download, or be usable without a network connection.
- Saving web application generated documents locally for offline use

Storage on the Browser

- There are two types of storage locations:
  - local storage
    - mostly used for settings
    - it has a limit (at least 5M)
    - equivalent to Preferences in Android
  - databases
    - fully fledged databases that allow to store relevant amounts of information
      - Web Storage
      - Indexed DB
      - Caches



Step 3 - Creating Functions

Now we need to create functions that will be called when the buttons are tapped. First function is used for adding data to local storage.

```
function setLocalStorage() {
    localStorage.setItem("Name", "John");
    localStorage.setItem("Job", "Developer");
    localStorage.setItem("Project", "Cordova Project");
}
```

The next one will log data we added to console.

```
function showLocalStorage() {
    console.log(localStorage.getItem("Name"));
    console.log(localStorage.getItem("Job"));
    console.log(localStorage.getItem("Project"));
}
```


IndexedDB

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
https://developers.google.com/web/ilt/pwa/working-with-indexeddb

Note: indexedDB is a changing API. We will not use the base API - we will use a library built on it so to reduce the dependency on the shifting standard

Also the core API is verbose and rather obscure. However any library will still use the same core methods such as get and add. These are not documented in the libraries and therefore for those you will have to use the general documentation. So keep the links above.

IndexedDB

https://www.html5rocks.com/en/tutorials/offline/storage/

- A collection of "**object stores**" which you can just drop objects into
- Similar to noSQL Databases
  - no constraints on the object structure as it happens in relational databases
- It supports asynchronous and synchronous operations

Advantages:

https://www.html5rocks.com/en/tutorials/offline/storage/

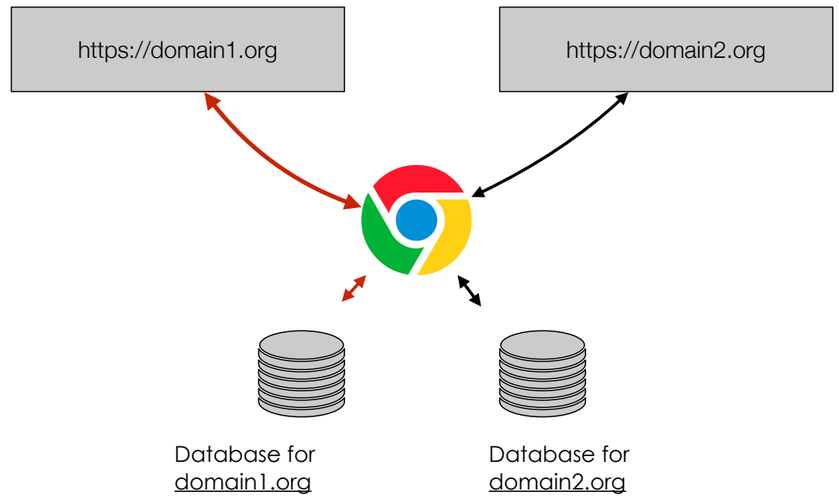
- Good performance (asynchronous API)
  - Database interaction won't lock up the user interface
- Good search performance,
  - data can be indexed according to search keys.
- Supports versioning
- Robust
  - it supports a transactional database model.
- Fairly easy learning curve
  - due to a simple data model
- Excellent browser support
  - Chrome, Firefox, mobile FF, IE10, Safari 10.1+, MS Edge

Getting Started

https://developers.google.com/web/ilt/pwa/working-with-indexeddb

- The pure form of IndexedDB is quite complex
  - We are using IndexedDB Promised library
    - https://github.com/jakearchibald/indexeddb-promised
  - It simplifies the API while maintaining its structure
- In IndexedDB each database is unique to an origin
  - the site domain or subdomain
  - it cannot access or be accessed by any other origin
- Its foundations are the object stores,
  - equivalent to relations in a relational database

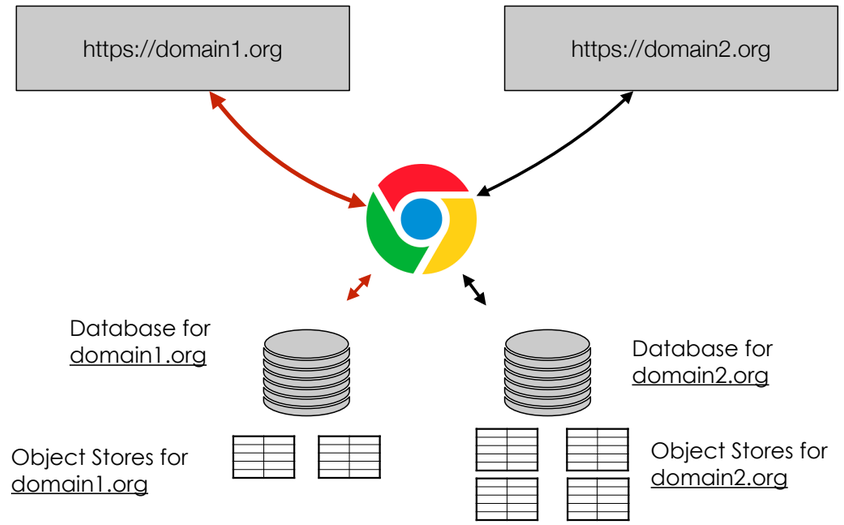
Data is Private to the Site



Terminology

- Database
  - contains the object stores
  - Multiple databases are possible with whatever names you choose
    - but try to limit to **one database per app**
- Object Store
  - An individual bucket to store data
    - equivalent to a relation in MySQL but IndexDB is a noSQL database
      - it stores Javascript objects rather than relations - see next week
  - Create one object store for each 'type' of data you are storing
    - For example, given an app that persists blog posts and user profiles, you could imagine two object stores (blog and users)
    - Unlike tables in traditional databases, the actual JavaScript data types within the store do not need to be consistent
      - you can store any Javascript object - but try to be consistent!

Data is Private to the Site



- Index
  - used for organising data in an object store by an individual property of the data
    - The index is used to retrieve records in the object store by this property
    - For example, if you're storing people, you may want to fetch them later by their name
- Transaction
  - A wrapper around an operation, or group of operations to ensure database integrity
  - If one of the actions fails, none of them are applied and the database returns to the previous state
  - Equivalent to transactions in relational databases
- Cursor
  - A mechanism for iterating over multiple records in database during search
    - equivalent to cursors in SQL databases

Checking for IndexedDB support

- Because IndexedDB isn't supported by all browsers, we need to check that the user's browser supports it before using it.
- The easiest way is to check the window object:

```
if (!('indexedDB' in window)) {
  console.log('This browser doesn\'t support IndexedDB');
  return;
}
```

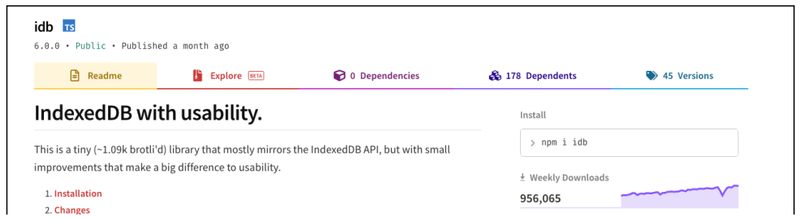
- (insert this instruction in your page initialisation method - the same place where you check if service-workers are provided)

in the lab…

```
 * showing any cached forecast data and declaring the service worker
 */
function initWeatherForecasts() {
    loadData();
    if ('serviceWorker' in navigator) {
        navigator.serviceWorker
            .register( scriptURL: './service-worker.js')
            .then(function () {
                console.log('Service Worker Registered');
            })
            .catch (function (error){
                console.log('Service Worker NOT Registered '+ error.message);
            });
    }
    //check for support
    if ('indexedDB' in window) {
        initDatabase();
    }
    else {
        console.log('This browser doesn\'t support IndexedDB');
    }
}
```


Using the idb library

- IndexedDB per se is quite cumbersome and wordy so we will use the idb library
  - https://www.npmjs.com/package/idb
- To install:
  - **if you want to use server side,** open the Terminal window in WebStorm and type
    - *npm i idb* (which is equivalent to *npm install idb*)
  - **if you want to use it via browser**, you have to link to a module (we will see how to use modules in the lab class)
    - *import * from 'https://unpkg.com/idb?module';*



idb as a Javascript module

- As idb is defined as a Javascript module, you include its functions as:

```
import { openDB, deleteDB, wrap, unwrap } from 'idb';
```

- if you have never used Javascript modules, you should look into them. They are more or less equivalent to classes of methods
  - I will go into some details of using modules in the lab class
- In idb it is suggested to use the construct async..await
  - see week 2
  - reminder: async… await is syntactic sugar for promises
    - you define an async function and use await in front of promises

```
async function accessDatabaseFunction() {
    const db = await idb.openDB(…);
  }
```


Opening a database

With IndexedDB you can create multiple databases with any names you choose. In general, there is just one database per app. To open a database, we use:

```
idb.open(name, version, upgradeCallback)
```

This method returns a promise that resolves to a database object. When using `idb.open`, you provide a name, version number, and an optional callback to set up the database.

Here is an example of `idb.open` in context:

```
const db = await openDB(name, version, {
  upgrade(db, oldVersion, newVersion, transaction) {
    // …
  },
  blocked() {                          called when it is not possible to connect
    // …
  },                                   called when it is not possible to connect
  blocking() {
    // …
  },
  terminated() {                       called when it is not possible to connect
    // …
  },
});
```

Opening (creating) and deleting

`openDB(name, version, { upgrade(db, oldVersion, newVersion, transaction)….})`

version (optional): Schema version, or undefined to open the current version.

`upgrade` (optional): Called if this version of the database has never been opened before. Use it to specify the schema for the database.

- `db`: a database.
- `oldVersion`: Last version of the database opened by the user.
- `newVersion`: Whatever new version you provided.
- `transaction`: An enhanced transaction for this upgrade. This is useful if you need to get data from other stores as part of a migration.

`deleteDB`

Deletes a database.

```
await deleteDB(name, {
  blocked() {
    // …
  },
});
```

- `name`: Name of the database.
- `blocked` (optional): Called if the database already exists and there are open connections that don't close in response to a versionchange event, the req will be blocked until they all close.

Creating object stores

- A database typically contains one or more object stores
- To ensure database integrity
  - object stores can only be created and removed in the callback function in the open function call
  - The callback receives an instance of UpgradeDB, a special object in the IDB Promised library that is used to create object stores
  - Call the createObjectStore method on UpgradeDB to create the object store

Example

```
async function demo() {
  const db = await openDB('Articles', 1, {
    upgrade(db) {
      // Create a store of objects if it does not exist
      const store = db.createObjectStore('articles') );
```

`createObjectStore(name, options);`
creates the equivalent of a relation in a relational database, i.e. a group of objects containing comparable data (e.g. users, devices, photos, etc.)

(In the MongoDB lecture we will see how noSQL databases work)

Defining primary keys

- to define how data is uniquely identified in the store

```
db.createObjectStore('people', {keyPath: 'email'});
```

- the key path is a property in the Javascript object stored in the object store
  - IndexedDb is a schema-less db
    - i.e. elements are not required to have a pre-defined structure
    - however if a key path is defined every object must have the property
- You can use auto increments

```
db.createObjectStore('logs', {keyPath: 'id', autoIncrement:true});
```

Defining indexes

- Indexes are defined for retrieval in object stores
  - used to retrieve data from the reference object store via a specified property

```
objectStore.createIndex('indexName', 'property', options);
```


Example

confused by the `(function…)()` wrapping?
https://www.tutorialspoint.com/Why-are-parenthesis-used-to-wrap-a-JavaScript-function-call
it basically creates a protected namespace

```
(function() {
  'use strict';

  //check for support
  if (!('indexedDB' in window)) {
    console.log('This browser doesn\'t support IndexedDB');
    return;
  }

  var dbPromise = idb.open('test-db4', 1, function(upgradeDb) {
    if (!upgradeDb.objectStoreNames.contains('people')) {
      var peopleOS = upgradeDb.createObjectStore('people', {keyPath: 'email'});
      peopleOS.createIndex('gender', 'gender', {unique: false});
      peopleOS.createIndex('ssn', 'ssn', {unique: true});
    }
•(…)
```

name of the index

field in the JS Object

options

Processing Data

- how to create, read, update, and delete data
  - they are async operations
- All data operations in IndexedDB are carried out inside a transaction
  - Each operation is organised as:
    - Get database object
    - Open transaction on database
    - Open object store on transaction
    - Perform operation on object store
    - you do not need to close a transaction but a promise is called to inform you when done in case it is useful
      - transaction.complete

Adding data to an object store

Creating data

To create data, call the add method on the object store and pass in the data you want to add.

Add has an optional second argument that lets you define the primary key for the individual object on creation

but it should only be used if you have not specified the key path in `createObjectStore`

```
transaction.add(data, optionalKey);
```

The data parameter can be data of any type: a string, number, object, array, and so forth.

The only restriction is if the object store has a defined keypath, the data must contain this property and the value must be unique

The add method returns a promise that resolves once the object has been added to the store.

```
const tx = db.transaction('articles', 'readwrite');
tx.store.add({
        title: 'Article 2',
        date: new Date('2019-01-01'),
        body: '…',
      }),
```

Note: adding to noSQL

- Indexed DB is a type of noSQL database, i.e. non relational
- In a NoSQL database, you can add any object, whatever its structure
  - as long as it has the appropriate fields for the index and key

Promise.addAll

- it is possible to add a number of elements to database in one go

```
async function demo() {
  const db = await openDB('Articles', 1, {…});
  const tx = db.transaction('articles', 'readwrite');
  await Promise.all([
       tx.store.add({
         title: 'Article 2',
         date: new Date('2019-01-01'),
         body: '…',
       }),
       tx.store.add({
         title: 'Article 3',
         date: new Date('2019-01-02'),
         body: '…',
       }),
       tx.done,
     ]);
  }
…
```

- Note: no need to close the transaction

Important! Always check that the transaction completed successfully by capturing the output of transaction complete (tx.done)!!

tx.done Vs tx.complete

- Transaction.complete https://github.com/voorhoede/code-class-indexeddb#transactioncomplete
  - When the transaction was carried out, transaction.complete will resolve, and reject if it failed.
    - In my experience, this is hardly ever useful and in some cases, will return undefined whether something happened or not.
- Transaction.done https://github.com/jakearchibald/idb
  - Transactions have a .done promise which resolves when the transaction completes successfully, and otherwise rejects with the transaction error

```
const tx = db.transaction(storeName, 'readwrite');
await Promise.all([
  tx.store.put('bar', 'foo'),
  tx.store.put('world', 'hello'),
  tx.done,
]);
```

- In the lab classes I may use .complete. It is better to use.done

Reading data

To read data, call the `get` method on the object store.

The `get` method takes the primary key of the object you want to retrieve from the store. Here is a basic example:

```
transaction.get(primaryKey);
or
index.get(conditions);
```

As with `add`, the `get` method returns a promise and must happen within a transaction.

Let's look at an example of the `get` method:

```
dbPromise.then(function(db) {
  var tx = db.transaction('store', 'readonly');
  return tx.get('sandwich');
}).then(function(val) {
  console.dir(val);    // this is not suggested
                       // if you have an index, use index})
```

```
async function demo() {
    const db = await openDB('Articles', 1, {
     upgrade(db) {
       // Create a store of objects if it does not exist
       const store = db.createObjectStore('articles') );
     });
    const tx = await db.transaction('articles', 'readwrite');
    // here we use the transaction but if you have an index it is more
    // efficient to use the index, see next slide
    var store = await tx.objectStore('articles');
    const mySandwich= await store.get('sandwich');
    console.log('your sandwich is: ' mySandwich.sandwich);
  }
```


Use Indexes if possible

- Suppose we want to check if login/password are in a database

```
async function getLoginData(loginObject) {
   if (dbPromise) {
    console.log('fetching: '+login);
    var tx = await db.transaction(LOGIN_STORE_NAME, 'readonly');
     var store = await tx.objectStore(LOGIN_STORE_NAME);
     var index = await store.index('userId');
     let foundObject= await index.get(IDBKeyRange.only(loginObject.userId));
    if (foundObject && (foundObject.userId==loginObject.userId && foundObject.password==loginObject.password){
              console.log("login successful");
           } else {
              alert("login or password incorrect")
           }
       });
    }
}
```

- ***IDBKeyRange***.*only* checks for equality of an index in a database

Updating data

To **update** data, call the `put` method on the object store.

The `put` method is very similar to the `add` method and can be used instead of `add` to create data in the object store.

someObjectStore.put(data, optionalKey);

Again, this method returns a promise and occurs inside a transaction. As with `add`, we need to be careful to check `transaction.complete` if we want to be sure that the operation was actually carried out.

```
// I have omitted the declaration of the variable db and the declaration of the async function
// see how we did it in the previous slides
var item = {
     name: 'sandwich',
     price: 4.99,
     description: 'A very tasty sandwich',
     created: new Date().getTime()
   };
try{
    var tx = await db.transaction('store', 'readwrite');
    var store = await tx.objectStore('store');
    await store.put(item);  // necessary as it returns a promise
    await tx.complete;
    console.log('added item '+ item.name + 'to the store');
} catch(error){
   console.log('failed to add item '+ item.name+' error:' + error); }
```


Deleting data

To delete data, call the `delete` method on the object store.

```
someObjectStore.delete(primaryKey);
```

Once again, this method returns a promise and must be wrapped in a transaction

```
async function deleteItem(db, storeName, key){
  var tx = await db.transaction(storeName, 'readwrite');
  var store = await tx.objectStore(storeName);
  await store.delete(key);
  await tx.complete;
  console.log('Item deleted');
 // as per the previous example you may consider checking for error with
 // try/catch
 }
```

Searching

```
someObjectStore.getAll(optionalConstraint);
```

This method returns all the objects in the object store matching the specified key or key range or all objects in the store if no parameter is given.

```
async function deleteItem(db, storeName, key){
  var tx = await db.transaction('store', 'readonly');
  var store = await tx.objectStore('store');
  let items= await store.getAll();
  await tx.complete;
  console.log('Items by name:', items);
};
```

Asking for all objects in a db is generally a bad idea

Searching using cursors

- A cursor selects each object in an object store or index one by one, letting you do something with the data as it is selected
- We create the cursor by calling the openCursor method on the object store

```
someObjectStore.openCursor(optionalKeyRange, optionalDirection);
```

- This method returns a promise that resolves with a cursor object representing the first object in the object store (or undefined)
- To move on to the next object in the object store, we call `cursor.continue`


```
async function getElements(db, storeName){
  let tx = await db.transaction(storeName, 'readonly');
  let store = await tx.objectStore(storeName);
  let cursor= await store.openCursor();
  if (!cursor) {
    return;
  }
  console.log('Cursored at:', cursor.key);
  for (var field in cursor.value) {
    console.log(cursor.value[field]);
  }
  await cursor.continue().then(logItems);
 await tx.complete;
  console.log('Done cursoring');
});
```


Query by range

see all the possible range conditions at https://developer.mozilla.org/en-US/docs/Web/API/IDBKeyRange

```
await function searchItems(db, storeName, lower, upper) {
 if (!db) return;
 if (lower === '' && upper === '') {return;}

 // build the range object
  let range;
  if (lower !== '' && upper !== '') {
    range = IDBKeyRange.bound(lower, upper);
  } else if (lower === '') {
    range = IDBKeyRange.upperBound(upper);
  } else {
    range = IDBKeyRange.lowerBound(lower);
  }

    let tx = await db.transaction([storeName], 'readonly');
    let store = await tx.objectStore(storeName);
    let index = await store.index('price');
  // use the range object. NOTE!! as query an index, you must open the cursor on THE
  // INDEX rather than on the STORE!!!
      let cursor= await index.openCursor(range);
    while (cursor){
     console.log('Cursored at:', cursor.key);
     for (var field in cursor.value)
       console.log(cursor.value[field]);
     await cursor.continue()
    }
  await tx.complete;
 }
```

Query by single value

- Suppose you want to query an index for a single value
  - use ***IDBKeyRange.only***(value) in the condition
  - this time we will use getAll rather than the cursor
    - use getAll if you want to get a list of all the items in theDB satisfying a specific condition

```
async function getDataAboutAValue(db, storeName, aValue) {
    if (!db) return;
            console.log('fetching: '+aValue+ ' from database');
            var tx = await db.transaction(storeName, 'readonly');
            var store = await tx.objectStore(storeName);
            var index = await store.index('someindex');
            let itemsList= await index.getAll(IDBKeyRange.only(aValue));
            if (itemsList && itemsList.length>0){
                  // do something with the items…
            } …
```


Versioning

new version

```
(async () => {
  //...
  const dbName = 'mydbname'
  const storeName = 'store0'
  const version = 1

  const db = await openDB(dbName, version, {
    upgrade(db, oldVersion, newVersion, transaction) {
      switch (oldVersion) {
        case 0: // no db created before
          // a store introduced in version 1
          db.createObjectStore('store1')
        case 1:
          // a new store in version 2
          db.createObjectStore('store2', { keyPath: 'name' })
      }
      db.createObjectStore(storeName)
    }
  })
})()
```

Used to upgrade the database to a new version – it keeps data consistent!!

Summary

- You should understand that you can create a rather large database using Javascript
- We have mainly focussed on the browser side
  - where you can build a db up to 1Gbyte for your site
- You should be able to
  - create an Indexed DB
  - add stores (i.e. the equivalent of relations)
  - create indexes
  - create and execute transactions
  - add, delete, put and search items
- The lab class will focus on the practicality of doing so

Questions?

Next lab is about IndexedDB

Service Workers

The backbone supporting the offline experience in PWAs

Prof. Fabio Ciravegna
Department of Computer Science
The University of Sheffield
f.ciravegna@shef.ac.uk

P.S. if I ever write or say Web Worker, I mean Service Worker!!

Service Workers

https://developers.google.com/web/fundamentals/primers/service-workers/

- A service worker is a script that your browser runs in the background,
  - separate from a web page,
  - opening the door to features that don't need a web page or user interaction.
- Today, they already include features like push notifications and background sync
  - In the future, service workers might support other things like periodic sync or geofencing
- The reason this is such an exciting API is that it allows you to support offline experiences, giving developers complete control over the experience

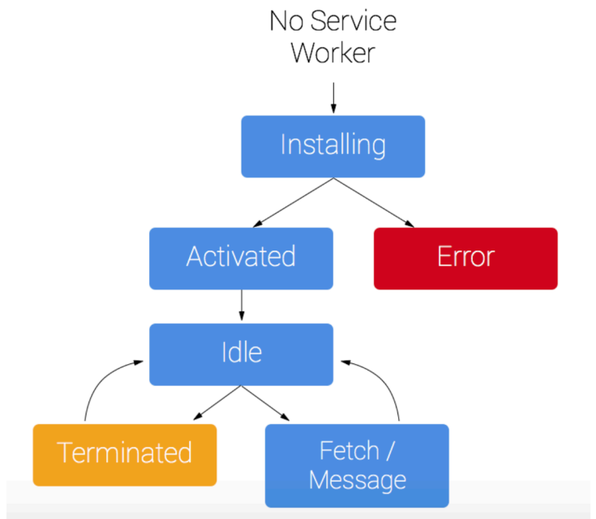
Features of SW

- It is a programmable **network** proxy
  - allowing to control that network requests from your page are handled (so it can be seen as an evolution of the Ajax paradigm)
- It's terminated when not in use, and restarted when it's next needed
  - you cannot rely on global state
  - If there is information that needs persisting and reusing across restarts, save the status using the IndexedDB API
- It's a JavaScript Worker,
  - so it can't access a page's DOM directly
  - A service worker can communicate with the pages it controls via messages and communicate with pages which will manipulate the DOM if needed
- Service workers make extensive use of promises
  - make sure to understand them!

Life Cycle

- Their lifecycle is completely separate from your web page
  - they are not just a Javascript script running on your page, they have an independent life and can run in the background
    - for example to provide you notifications even when you are not on the page
  - they are started by running a JS script in a page but then they take their own life
    - think of a background process in mobile computing
      - they are launched by the UI Thread and then become independent and can only communicate with the UI Thread via a specific communication channel
- Steps:
  - installation
    - cached assets (web pages) are loaded
  - activation
    - the service worker controls network communication between the pages and the server

Life Cycle of a Service Worker



Requirements

- Browser support:
  - Latest versions of browsers fully support them
    - IE is not a browser
- HTTPS Secure connection to the server
  - A service worker effectively hijacks connections, fabricates, and filters responses
    - potentially opening the door to dangerous man-in-the-middle attacks
    - you can only register service workers on pages served over HTTPS,
      - so we know the service worker the browser receives hasn't been tampered with during its journey through the network

Life Cycle

- To register a service worker, you must execute a script on the website's home page

```
// if the browser supports service workers

if ('serviceWorker' in navigator) {
 // run this code at loading time
  window.addEventListener('load', function() {
    // register the service worker. the service worker is a js file
    // AND *must* be installed on the root of the set of pages it
    // controls. You can reload the pages as many times as you want
    // do not worry about registering a service worker multiple times
     navigator.serviceWorker.register('/sw.js')
      .then(function(registration) {
         // Registration was successful
        console.log('ServiceWorker registration successful with scope:',
         registration.scope);
      }, function(err) {
         // registration failed :(
          console.log('ServiceWorker registration failed: ', err);
      });
  });}
```

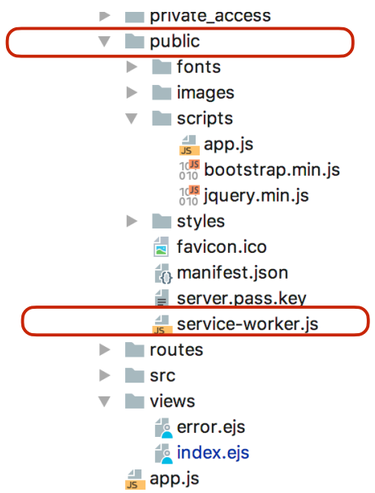

Scope of a SW

- The register() method shows the location of the service worker file
- The service worker's scope is the part of the file system controlled by its location
  - e.g. in the following example, the SW controls folder 2 and its subfolders 2.1 and 2.2
    - root
      - folder 1
      - subfolder 1.1
      - subfolder 1.2
      - folder 2
      - service worker
      - subfolder 2.1
      - subfolder 2.2

Scope of service worker

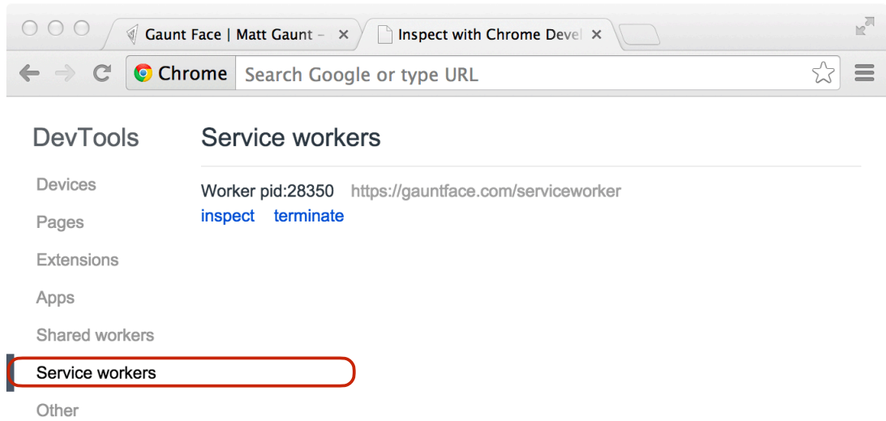
- In other words, this service worker will receive fetch events for everything under folder 2 (inclusive)

- If you want the sw to control the entire site, register it at the root of your public directory
  - NOT in the views directory
  - as the worker is a static file and hence it must be in the public directory

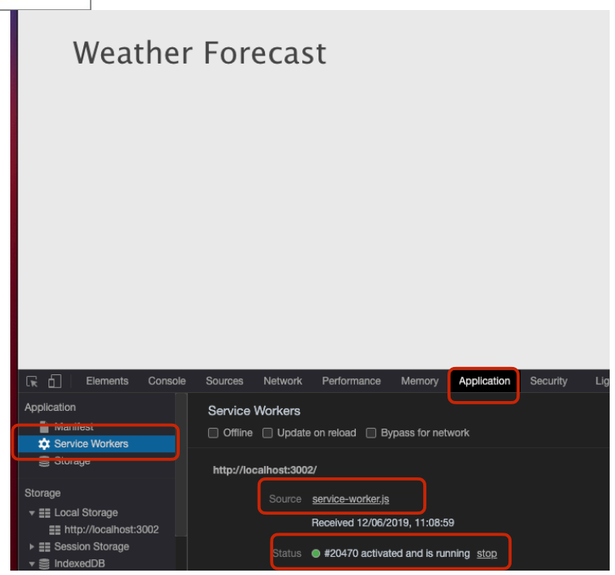


See it

- chrome://inspect/#service-workers



Much better: Chrome's debugger



Installing a SW

- The typical use of a SW is to cache files e.g. to allow for a smooth off-line experience.
- Steps:
  - Open its cache(s)
  - Cache your files
  - Confirm whether all the required assets are cached or not
- The SW is a js file that is loaded by the registration

```
navigator.serviceWorker.register('/sw.js')
```

- In the sw.js file you must
  - define a callback for the install event

code in sw.js

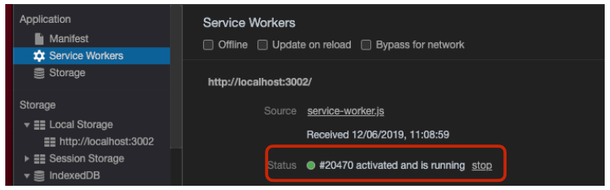
```
var CACHE_NAME = 'my-site-cache-v1';
var urlsToCache = [
 // add the files you want to cache here - no need to cache the service worker
  '/',
  '/styles/main.css',
  '/script/main.js'
];

self.addEventListener('install', function(event) {
  // Perform install steps
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(function(cache) {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );
});
//event.waitUntil() receives a promise as parameter
// and waits until the promise is fulfilled
// it returns if it succeeded or not
```


Careful here

- If any of the files to be cached fails to load,
  - none of the files will be cached
  - the service worker will not start
  - your pages and services will not be available off-line
  - So be very careful about providing a long list of files
    - as if the loading breaks, your service worker will fail
- When you declare a service worker for the first time,
  - the service worker will become available only **after** the user navigates to a different page or refreshes the current page
  - not before!!!
  - this may be confusing sometimes

Also

- Be careful: after caching the pages, the service worker will always return that version of the page without going to the server ever again
  - that means it will never receive any updates unless you force it
  - We will see different situations and how to avoid issues
  - But if during debugging it appears that your changes are not shown/the behaviour of your programme does not work
    - it is very possible that it is the caching done by your service worker
    - in that case kill the service worker using Chrome's debugger (see image)



Fetch event

- When the browser tries to load a page from a site that is controlled by the Service worker
  - a fetch event is fired by the browser
  - the fetch event is captured by the service worker which may decide to serve the cached version of the page instead of fetching it from the network
  - this is the natural continuation of the Ajax idea
    - just go to the server when necessary
      - only download data that is strictly necessary/not already available locally
    - the difference with Ajax is that the worker will act automatically on every fetch, without having to call it explicitly as we do with Ajax

```
// when the worker receives a fetch request
self.addEventListener('fetch',(event) => {
  event.respondWith(
  // it checks if the requested page is among the cached ones
     caches.match(event.request)
       .then(function(response) {
         // Cache hit - return the cache response (the cached page)
         if (response) {
           return response;
         } //cache does not have the page – go to the server
         return fetch(event.request); )  );});
```

Note: the event has a field representing the https request: the same you will receive on the nodejs side
- it has also the same parameters

Caching a retrieved page

- If the page is not among the cached one, the service worker may retrieve it from the server and cache it for you

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        // Cache hit - return response
        if (response) {
          return response;
        }
     // here we will cache the page – (see next slide)
```


ctd

```
// IMPORTANT: both requests and responses are streams and can only be consumed
// once. Since we are consuming them for both the cache and for the browser,
// we need to clone both the response and the request before using them for the
// second time
       var fetchRequest = event.request.clone();
       return fetch(fetchRequest).then(
         function(response) {
           // Check if we received a valid response. A basic response is one that
           // is made when we fetch from our own site. Do not cache responses to
           // requests made to other sites
           // if the file does not exist, do not cache - just return to the browser
           if(!response || response.status !== 200 || response.type !== 'basic') {
             return response;
           }
          // response is valid. Cache the fetched file
           // IMPORTANT: as mentioned we must clone the response.
           //  A response is a stream
           // and because we want the browser to consume the response
           // as well as the cache consuming the response, we need
           // to clone it so we have two streams.
           var responseToCache = response.clone();
           caches.open(CACHE_NAME)
             .then(function(cache) {
               cache.put(event.request, responseToCache); // here we use the clone
             });
           return response;  // here we use the original response
         }        );      })
```


That's it!

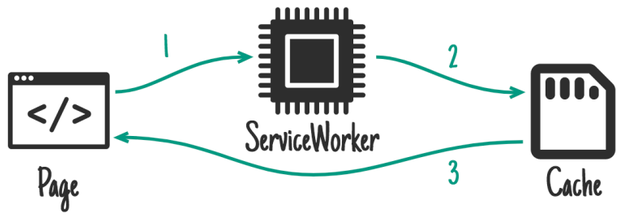
- Now we have a service worker that caches pages and can therefore provide our service even when the device is off-line
  - The user will be able to browse the site even when offline, offering the full experience similar to the mobile environment
- However… pages may change - sometimes they may change several times a day
  - How do we refresh the cache?

Caching Strategies for Service Workers

Caching strategies

- The strategy above is called *`Cache, falling back to Network`*
  - The browser will never be able to update the pages after cashing them
    - that is because in that code, we never check if the page has been updated
- In reality there are several strategies that we may need to implement depending on the requirements of the site
  - we will now see some of them

Cache only



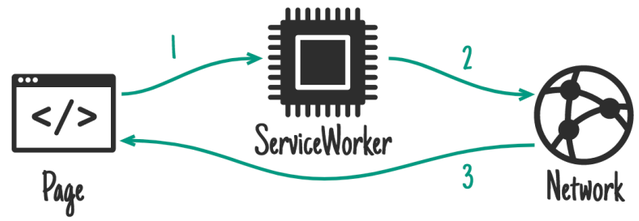
Ideal for: Anything you'd consider static to that "version" of your site. You should have cached these in the install event, so you can depend on them being there.

```
self.addEventListener('fetch', function(event) {
  // If a match isn't found in the cache, the response
  // will look like a connection error
  event.respondWith(caches.match(event.request));
});
```

…although you don't often need to handle this case specifically, "Cache, falling back to network" covers it.

https://jakearchibald.com/2014/offline-cookbook/#on-activate

Network only

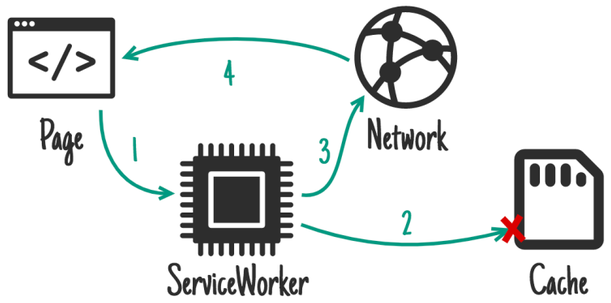


Ideal for: Things that have no offline equivalent, such as analytics pings, non-GET requests.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(fetch(event.request));
  // or simply don't call event.respondWith, which
  // will result in default browser behaviour
});
```

…although you don't often need to handle this case specifically, "Cache, falling back to network" covers it.

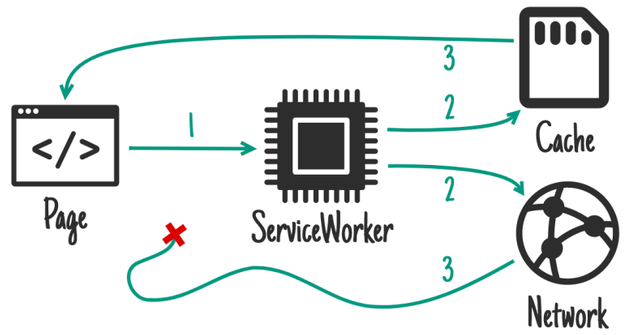
Cache, falling back to network



Ideal for: If you're building offline-first, this is how you'll handle the majority of requests. Other patterns will be exceptions based on the incoming request.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);
    })
```


Cache & network race

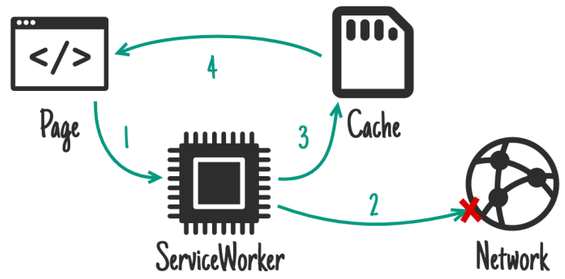


Ideal for: Small assets where you're chasing performance on devices with slow disk access.

With some combinations of older hard drives, virus scanners, and faster internet connections, getting resources from the network can be quicker than going to disk. However, going to the network when the user has the content on their device can be a waste of data, so bear that in mind.


```
// Promise.race is no good to us because it rejects if
// a promise rejects before fulfilling. Let's make a p
// race function:
function promiseAny(promises) {
  return new Promise((resolve, reject) => {
    // make sure promises are all promises
    promises = promises.map(p => Promise.resolve(p));
    // resolve this promise as soon as one resolves
    promises.forEach(p => p.then(resolve));
    // reject if all promises reject
    promises.reduce((a, b) => a.catch(() => b))
      .catch(() => reject(Error("All failed")));
  });
};

self.addEventListener('fetch', function(event) {
  event.respondWith(
    promiseAny([
      caches.match(event.request),
      fetch(event.request)
    ])
  );
});
```


Network falling back to cache



Ideal for: A quick-fix for resources that update frequently, outside of the "version" of the site. E.g. articles, avatars, social media timelines, game leader boards.

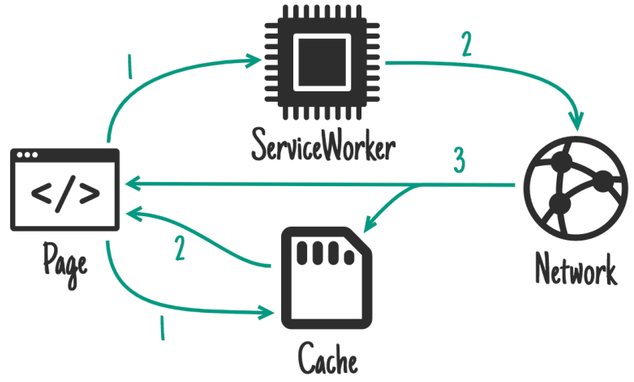
This means you give online users the most up-to-date content, but offline users get an older cached version. If the network request succeeds you'll most-likely want to update the cache entry.

However, this method has flaws. If the user has an intermittent or slow connection they'll have to wait for the network to fail before they get the perfectly acceptable content already on their device. This can take an

However, this method has flaws. If the user has an intermittent or slow connection they'll have to wait for the network to fail before they get the perfectly acceptable content already on their device. This can take an extremely long time and is a frustrating user experience. See the next pattern, "Cache then network", for a better solution.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request);
    })
  );
});
```

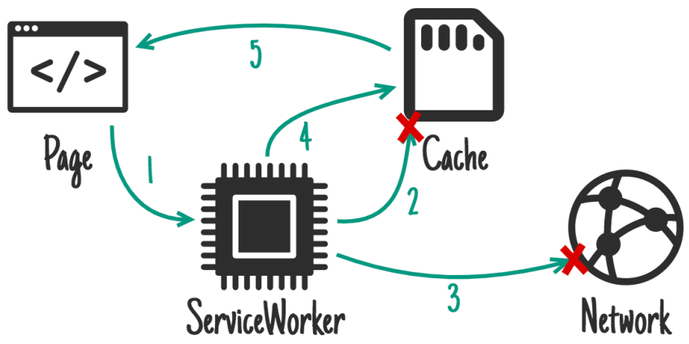

Cache then network



Ideal for: Content that updates frequently. E.g. articles, social media timelines, game leaderboards.

This requires the page to make two requests, one to the cache, one to the network. The idea is to show the cached data first, then update the page when/if the network data arrives.

Generic fallback



If you fail to serve something from the cache and/or network you may want to provide a generic fallback.

Ideal for: Secondary imagery such as avatars, failed POST requests, "Unavailable while offline" page.

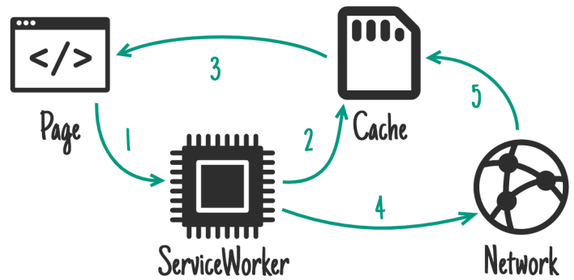

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    // Try the cache
    caches.match(event.request).then(function(response) {
      // Fall back to network
      return response || fetch(event.request);
    }).catch(function() {
      // If both fail, show a generic fallback:
      return caches.match('/offline.html');
      // However, in reality you'd have many different
      // fallbacks, depending on URL & headers.
      // Eg, a fallback silhouette image for avatars.
    })
  );
});
```

Sometimes you can just replace the current data when new data arrives (e.g. game leaderboard), but that can be disruptive with larger pieces of content. Basically, don't make disappear something the user may be reading or interacting with.

Twitter adds the new content above the old content & adjusts the scroll position so the user is uninterrupted. This is possible because Twitter mostly retains a mostly-linear order to content. I copied this pattern for trained-to-thrill to get content on screen as fast as possible, but still display up-to-date content once it arrives.

Stale-while-revalidate

Stale-while-revalidate



Ideal for: Frequently updating resources where having the very latest version is non-essential. Avatars can fall into this category.

If there's a cached version available, use it, but fetch an update for next time.

https://jakearchibald.com/2014/offline-cookbook/#on-activate

Stale-while-revalidate

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.match(event.request)
          .then(function(response) {
              var fetchPromise = fetch(event.request)
              .then(function(networkResponse) {
                  cache.put(event.request,
                            networkResponse.clone());
                return networkResponse;
              })
              return response || fetchPromise;
      })));
});
```


Summary

- You should understand the concept of Service Workers
  - in particular their role in the organisation of a PWA
- You should understand how they work
- You should be able to intercept requests for both GETs and a POSTs
- You should know how to organise your service worker so that you PWA has a consistent and appropriate behaviour with respect caching and retrieving from the server
  - ask yourself: what are the most important requirements of my PWA?
    - Is it the speed in serving results? Is it the recency of results?
    - what parts of the app/data are to be made available for offline use?
      - i.e. which parts do you cache?
    - what parts can be left as non working?
      - i.e. what parts are non cached?
    - What do you do with POSTs when you are offline?
      - do you tell the user they cannot post? Do you cache the results? (For the latter issue we will see IndexedDB next)

Questions?

https

- HTTPS protects the integrity of your website
  - HTTPS helps prevent intruders from tampering with the communications between your websites and your users' browsers.
    - Intruders include intentionally malicious attackers, and legitimate but intrusive companies, such as ISPs or hotels that inject ads into pages
- HTTPS protects the privacy and security of your users
  - HTTPS prevents intruders from being able to passively listen to communications between your websites and your users.
- HTTPS is the future of the web
  - Powerful, new web platform features require explicit permission from the user before executing
    - e.g. taking pictures or recording audio with getUserMedia(), (see lecture 6)
    - enabling offline app experiences with service workers, or building progressive web apps
- To add HTTPS to your server you'll need to get a TLS certificate and set it up for your server
  - This varies depending on your setup, so check your server's documentation and be sure to check out Mozilla's SSL config generator for best practices.

https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https

Working with HTTPS

Add https to nodeJs

- in bin/www
  - change

```
var http = require('http');
var server = http.createServer(app);
```

However in the lab you should not do this

  - into:

```
var https = require('https');
var options = {
      // the private key
    key: fs.readFileSync('./private/key.pem'),
      // The Certificate Signing Request (csr)
    cert: fs.readFileSync('./private/bundle.crt')
};
/**
 * Create HTTPs server using the options
 */
var server = https.createServer(options, app);
```

SSL connection

- An SSL connection between a client and server is set up by a handshake, the goals of which are:
  - To satisfy the client that it is talking to the right server (and optionally vice versa)
  - For the parties to have agreed on a "cipher suite", which includes which encryption algorithm they will use to exchange data
  - For the parties to have agreed on any necessary keys for this algorithm
- Once the connection is established, both parties can use the agreed algorithm and keys to securely send messages to each other

https://robertheaton.com/2014/03/27/how-does-https-actually-work/

- 3 main phases
  - Hello - The handshake begins with the client sending a ClientHello message.
    - This contains all the information the server needs in order to connect to the client via SSL, including the various cipher suites and maximum SSL version that it supports.
    - The server responds with a ServerHello, which contains similar information required by the client
  - Certificate Exchange - Now that contact has been established, the server has to prove its identity to the client
    - This is achieved using its SSL certificate, which is a very tiny bit like its passport
    - An SSL certificate contains various pieces of data, including the name of the owner, the property (eg. domain) it is attached to, the certificate's public key, the digital signature and information about the certificate's validity dates
    - The client checks that it either implicitly trusts the certificate, or that it is verified and trusted by one of several Certificate Authorities (CAs) that it also implicitly trusts.

- Key Exchange -
  - The encryption of the actual message data exchanged by the client and server will be done using a symmetric algorithm agreed during the Hello phase.
  - A symmetric algorithm uses a single key for both encryption and decryption, in contrast to asymmetric algorithms that require a public/private key pair.
  - Both parties need to agree on this single, symmetric key, a process that is accomplished securely using asymmetric encryption and the server's public/private keys.
- The client generates a random key to be used for the main, symmetric algorithm.
  - It encrypts it using an algorithm also agreed upon during the Hello phase, and the server's public key (found on its SSL certificate).
  - It sends this encrypted key to the server, where it is decrypted using the server's private key, and the interesting parts of the handshake are complete.
- The parties are sufficiently happy that they are talking to the right person, and have secretly agreed on a key to symmetrically encrypt the data that they are about to send each other. HTTP requests and responses can now be sent by forming a plaintext message and then encrypting and sending it.
- The other party is the only one who knows how to decrypt this message, and so Man In The Middle Attackers are unable to read or modify any requests that they may intercept.

Self signed certificates

- For debugging purposes you can generate a self signed certificate
  - NOT for public use
  - will not be trusted by the browsers
- Get an official certificate by a trusted provider for production (e.g. letsencrypt.org)

Self Signed Certificate

https://www.mobilefish.com/services/ssl_certificates/ssl_certificates.php

Certificate signing request, certificate and private key settings:

Country Name [C]*:	United Kingdom
State or Province Name (full name) [ST]*:	England
Locality Name (eg, city) [L]*:	Sheffield
Organization Name (eg, company) [O]:	The University of Sheffield
Organizational Unit Name (eg, section) [OU]:	Computer Science
Common Name (eg, your name, domain name or ip address) [CN]*:	localhost
Email Address:	f.ciravegna@shef.ac.uk
Protect the private key with a passphrase. Enter the passphrase:	whateverworksforyou
Characters entered:	19
RSA key bit length *:	2048
Number of days the certificate is valid *:	365
To prevent automated submissions an Access Code has been implemented for this tool.	Y0i
Please enter the Access Code as displayed above *:	Y0i

* = required

Generate | Clear all | Demo info