ASSIGNMENT 2

Progressive Web Apps
2020

You must create a Progressive Web App with the functionality described in this document.

The PWA will be hosted on a server provided for you and will be served via HTTPS , it must work offline, and be installable.

Features of Assessment:

Work Offline with Service Workers

Cache Management

Multithreaded with Web Workers

DOM Scripting

Web Services

Objects / Closures

Part 1: App Shell

Part 2: Search Flickr

Part3: Search Scripts

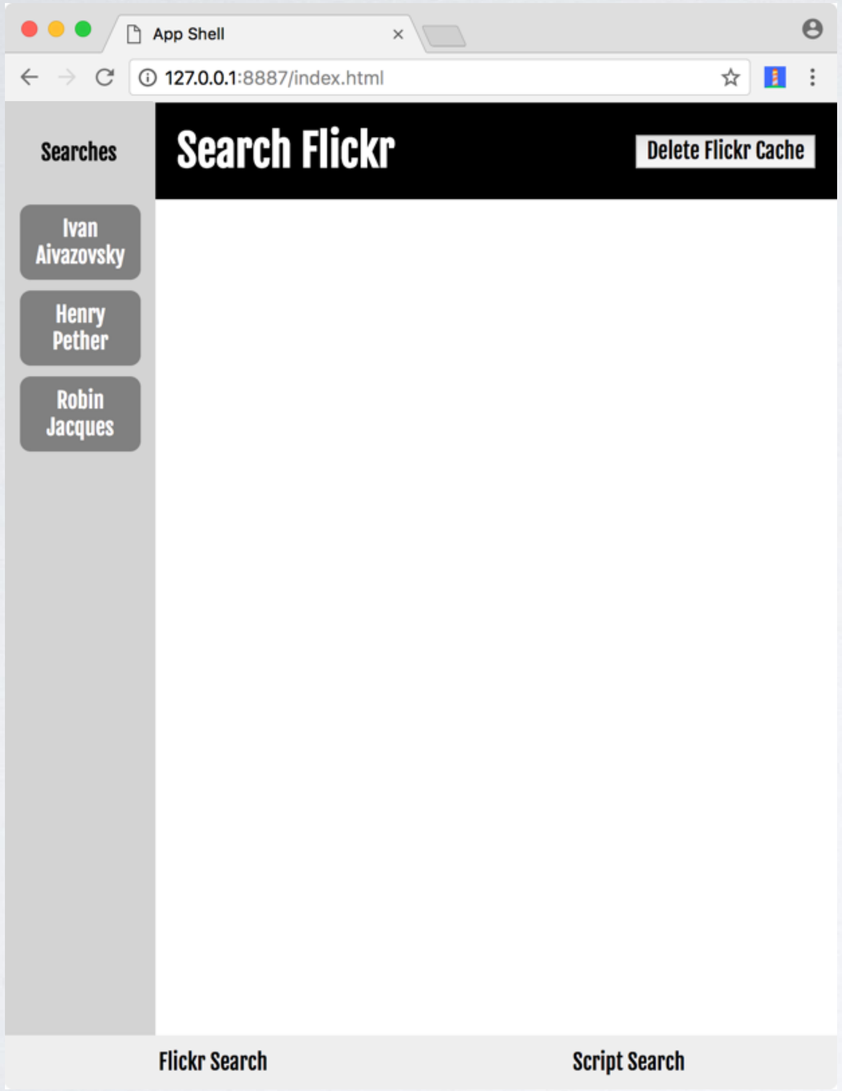
Part 4: Make your PWA Installable as app

Part 5: Caching Policies

Part 1: App Shell

Your app will consist of two pages/screens.

You must first create an **app shell** (based on your first assignment) that will be served from your cache, enabling your app to load if offline.



The links at the bottom of the page bring you to one of the 2 pages/screens.

Your app will have two main functions:

1) Search Flickr (using web service)

2) Search for Movie Scripts (using external JSON file)

These can be 2 separate pages.

Part 2: Search Flickr

Your page should have 3 buttons that initiate searches of Flickr.

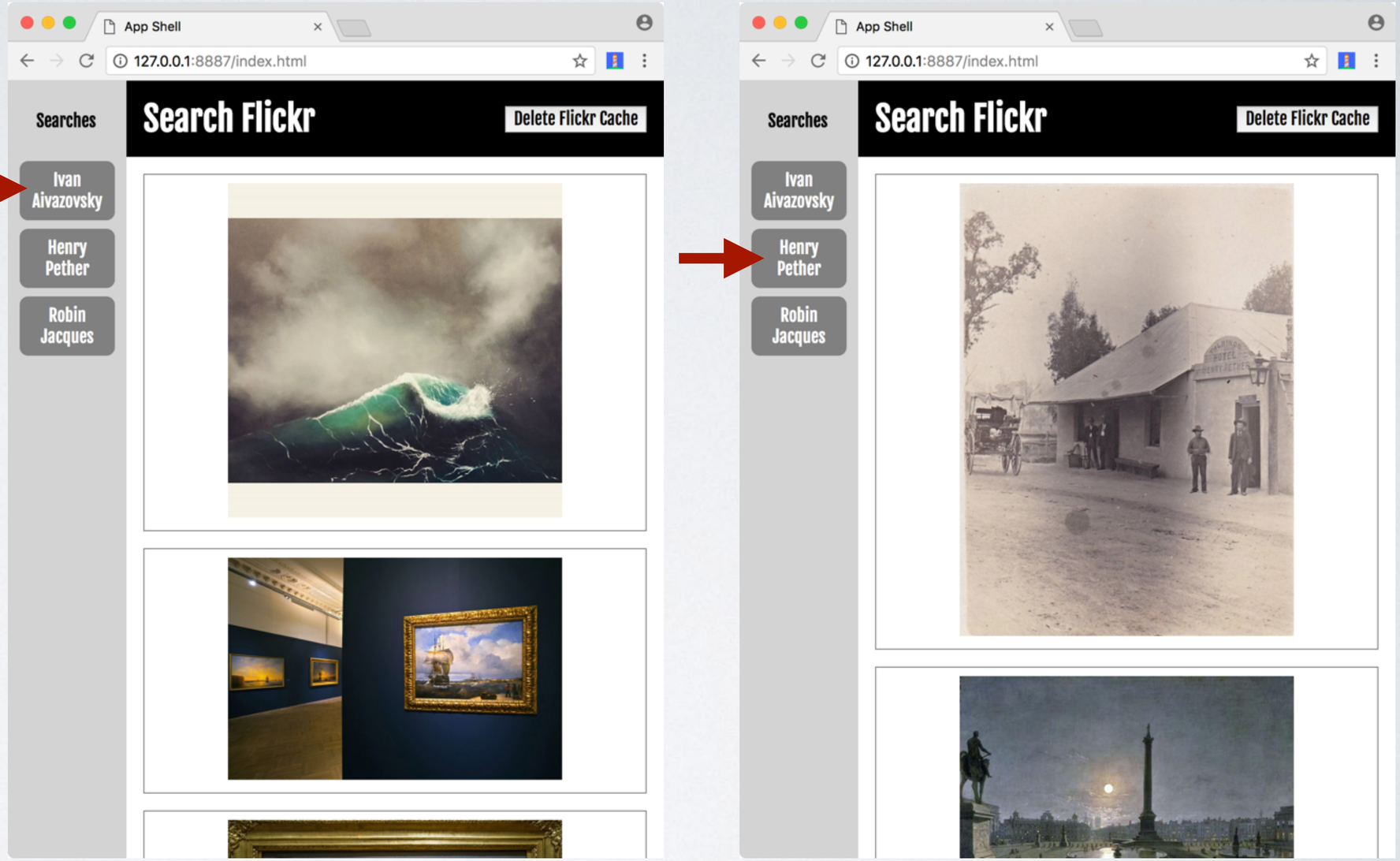
The search terms should be contained in an array and the buttons dynamically created from that array using DOM scripting.

E.g.

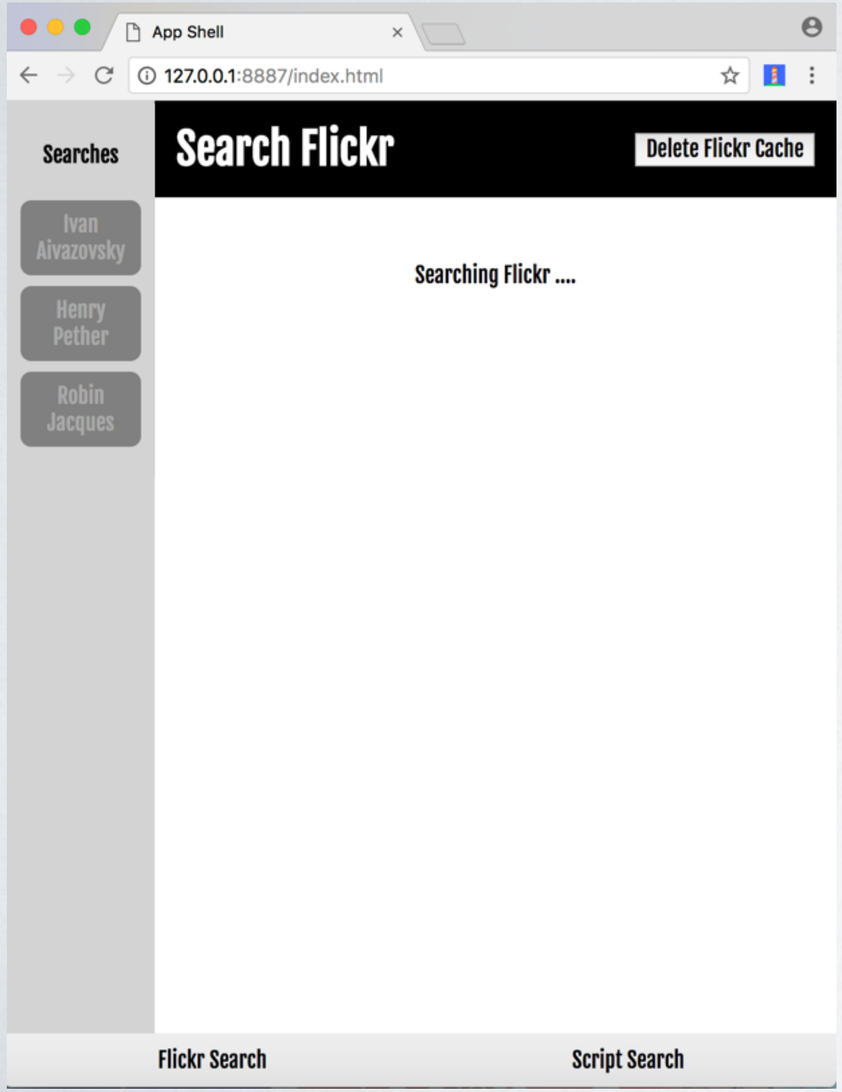
```
 var searchTerms = [
"Ivan Aivazovsky",
"Henry Pether",
"Robin Jacques"
];
```

If you click on one of the search terms you should search Flickr (with JSON-P) for that term and display the images that Flickr sends back (to a maximum of 10).

See the earlier Lab for information on searching Flickr.

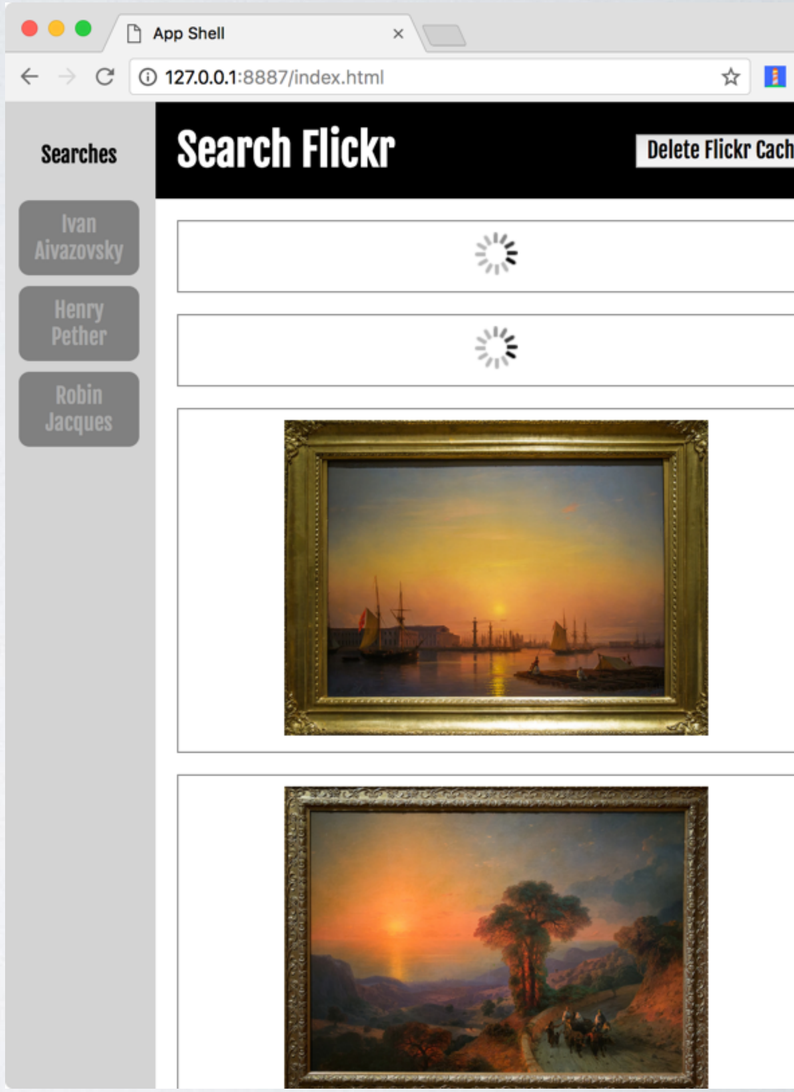


You should alert the user that the data is being retrieved.

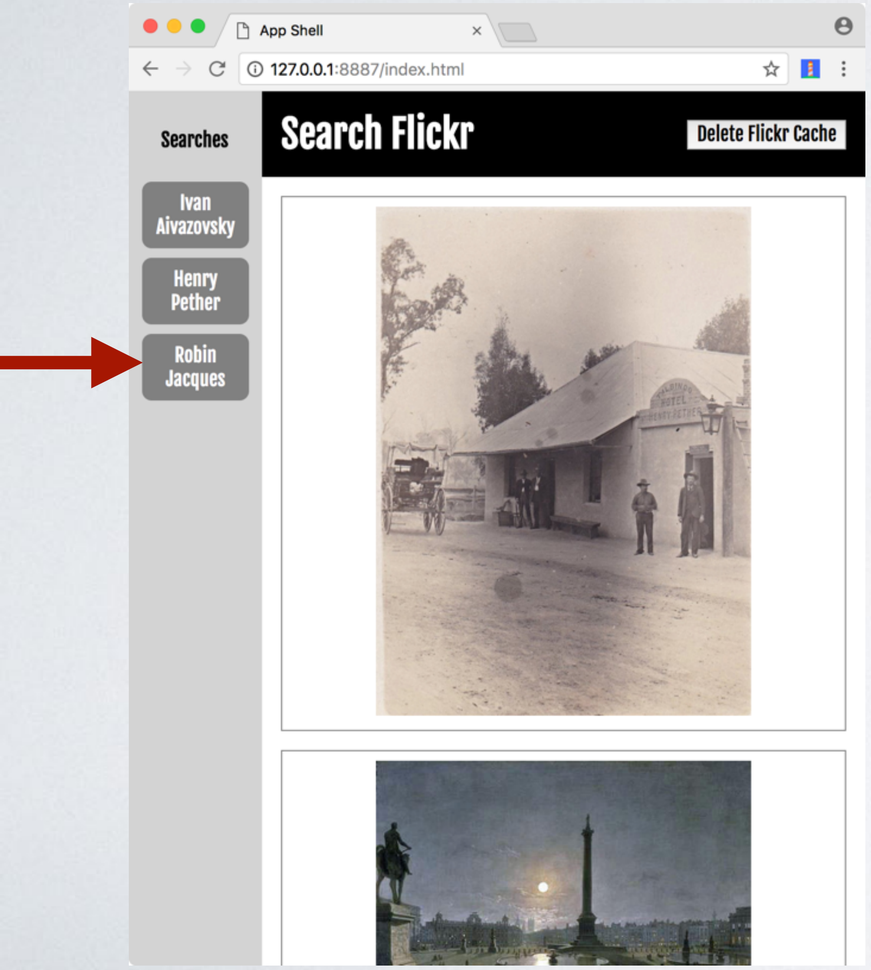


Once you generate the URLS. each image has a loading gif displayed until it downloads fully.

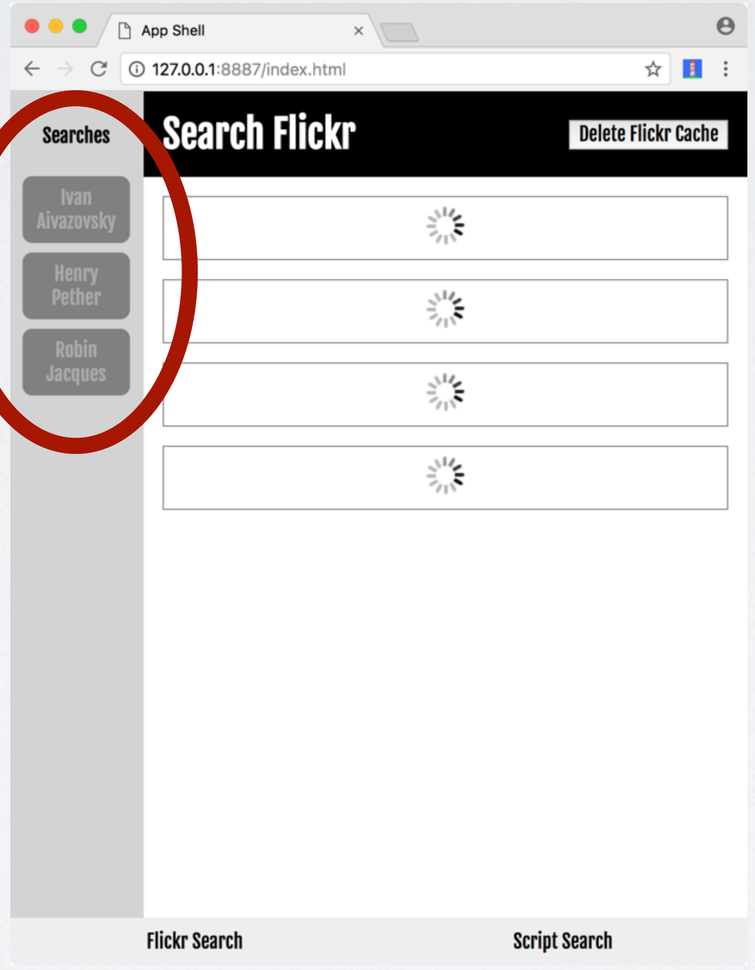
The loading gifs are eventually *individually* replaced by the *fully* downloaded images.



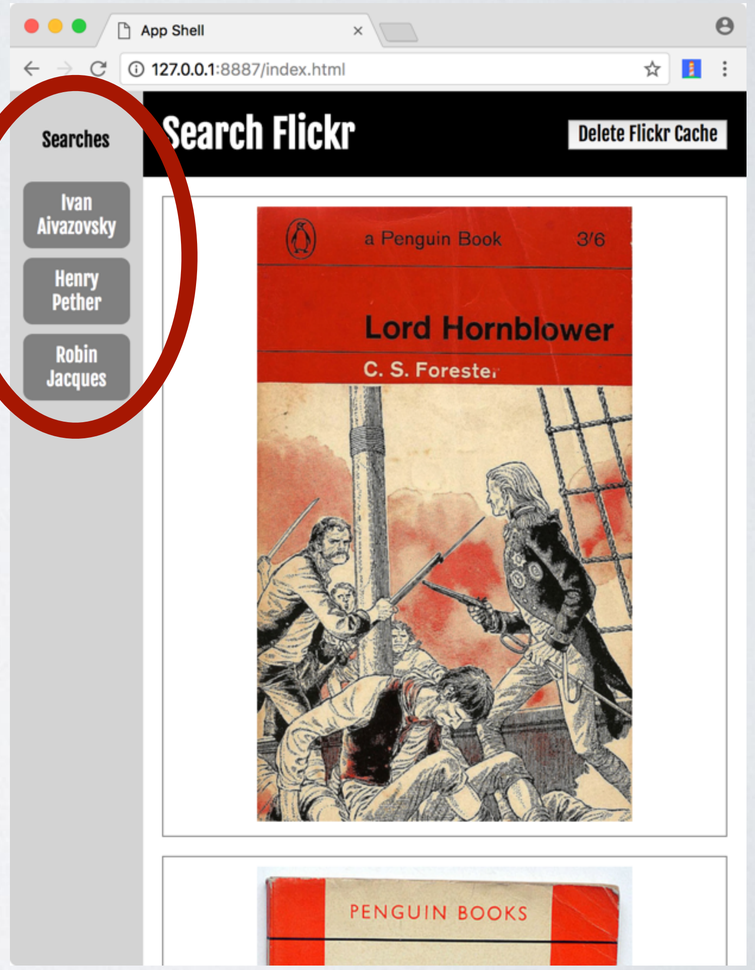
Note that the buttons are disabled when the searches started.



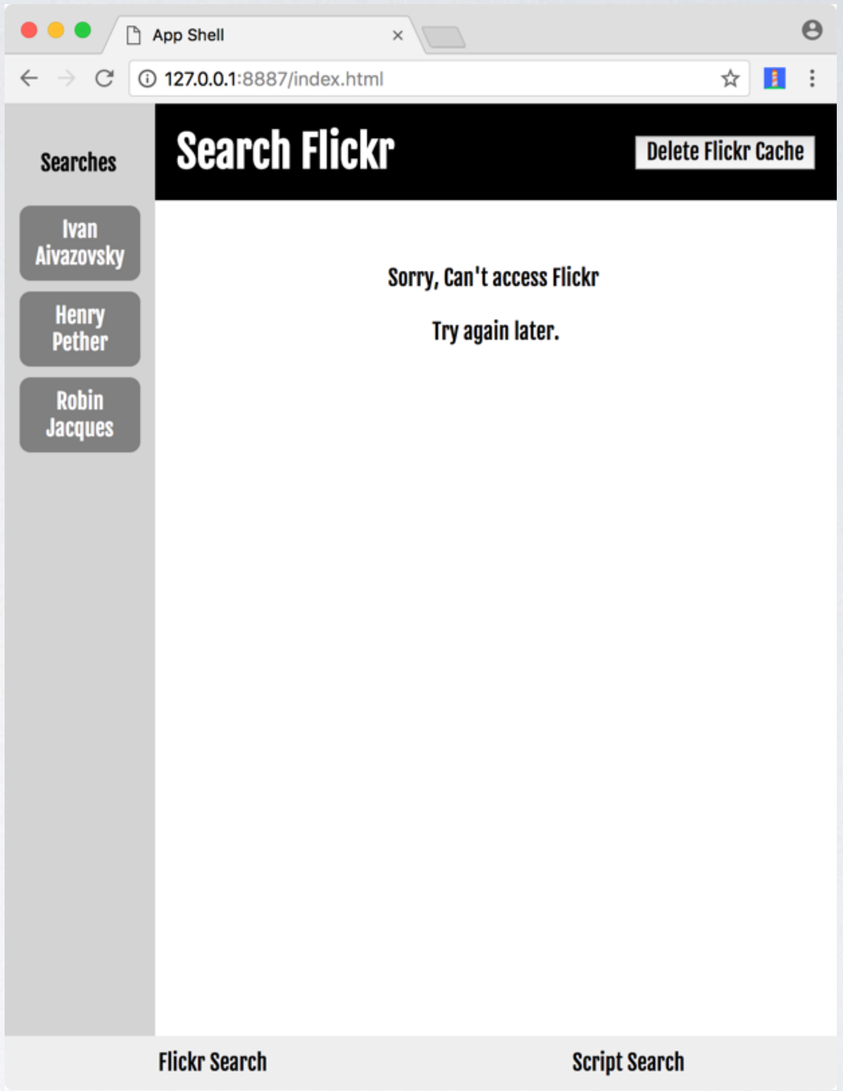
You must represent his visually somehow. The text is darkened in this example.



When **ALL** images have downloaded the buttons are reenabled.

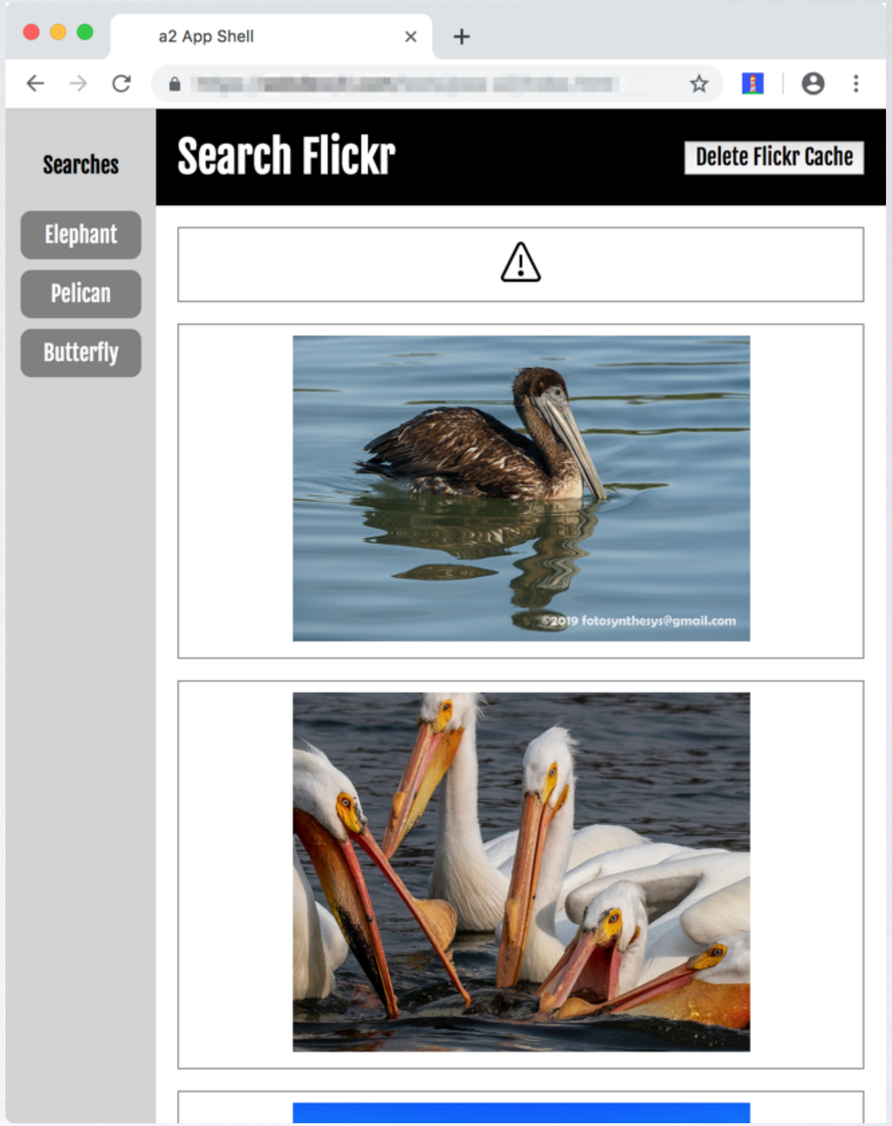


If you can't access Flickr (e.g. you are offline) you must let the user know gracefully.



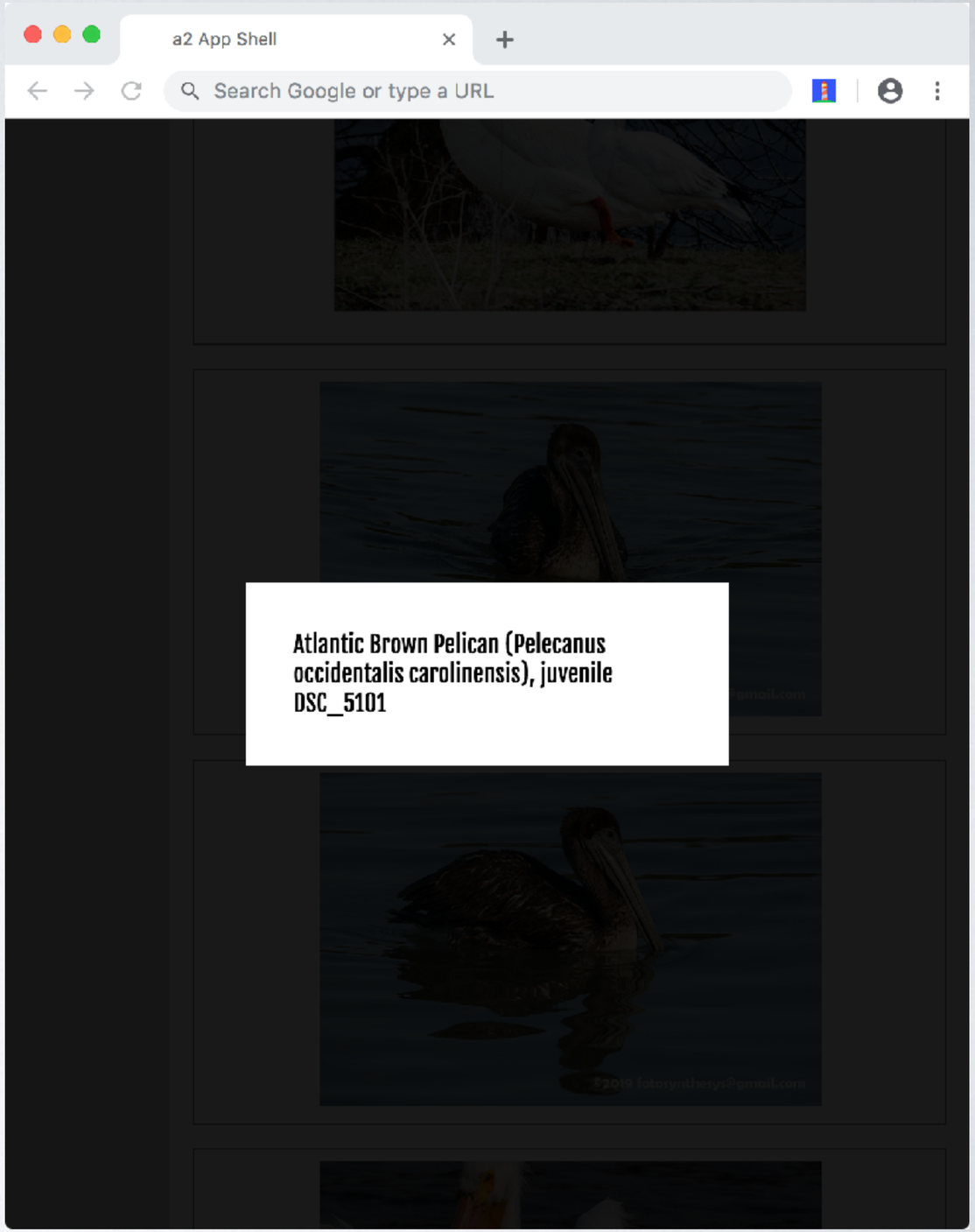
Similarly, your script should gracefully handle the cases where individual images don't load.

(I.e. they can't be accessed online and are not in the cache.)



Each image should have an **onclick** event handler so that when you click on the image it displays the title in a centered element that covers the entire window (as shown).

Clicking anywhere should hide the title and show the main page again (see screencast).



Note: You can use Promises to detect when individual images download. And Promises.all() to detect when they have all downloaded.

If an image doesn't download **it should still resolve** (i.e. **not reject**) so that you can re-enable the buttons. (I.e. this is important because Promise.all() won't resolve if any of the Promises you pass to it reject.)

localStorage

For the purposes of this assignment we will store data in the (blocking) **localStorage** API.

(Usually we would use the non-blocking IndexedDB.)

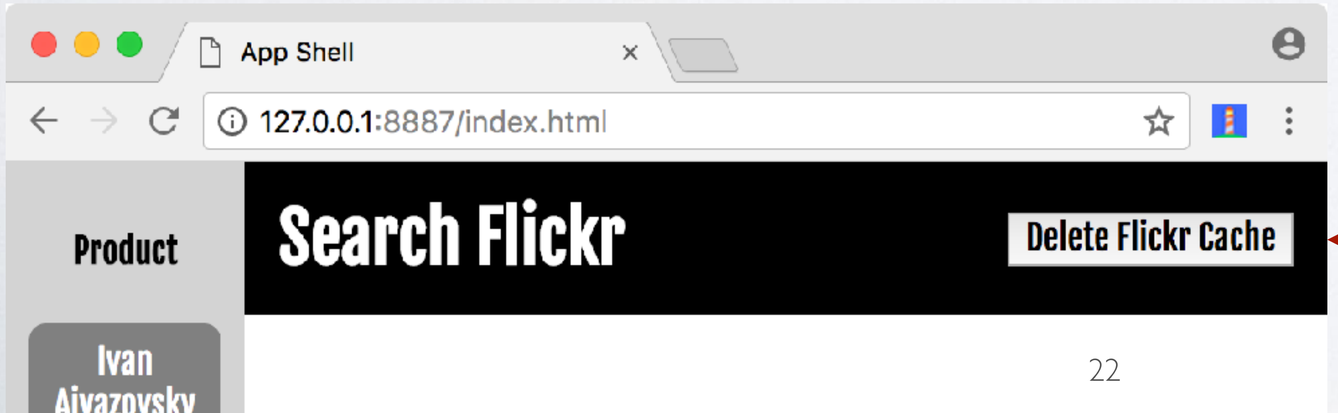
We will be storing an array of filenames of the last set of images we retrieved whenever we perform a search (i.e. we should always have the urls of the last search we performed in **localStorage**).

Since the images will be cached, we can then retrieve this array (if it is present), and add the images to the page immediately when we startup up the app (even if we are offline - since the images will be in the cache).

Managing the Cache

The Caching Policy for the app is discussed later.

But you must add a button to your page that allows you delete all the files you downloaded from Flickr (and ***only*** those files) from the cache. You shouldn't touch the other files such as those used for the app shell.



However, **you shouldn't delete the images of the last search** assuming there was one.

(You will know these images since you stored them in an array. i.e. we have the filenames stored in localStorage - See last section).

By leaving ***just*** these files in the cache there will be something on screen when users launch the app (assuming its not he first time they launched the app).

Implementation: Objects & Promises

To help implement loading the images from Flickr you should create a constructor function to create objects representing those images.

You should be able to pass the constructor a filename/url which it stores:

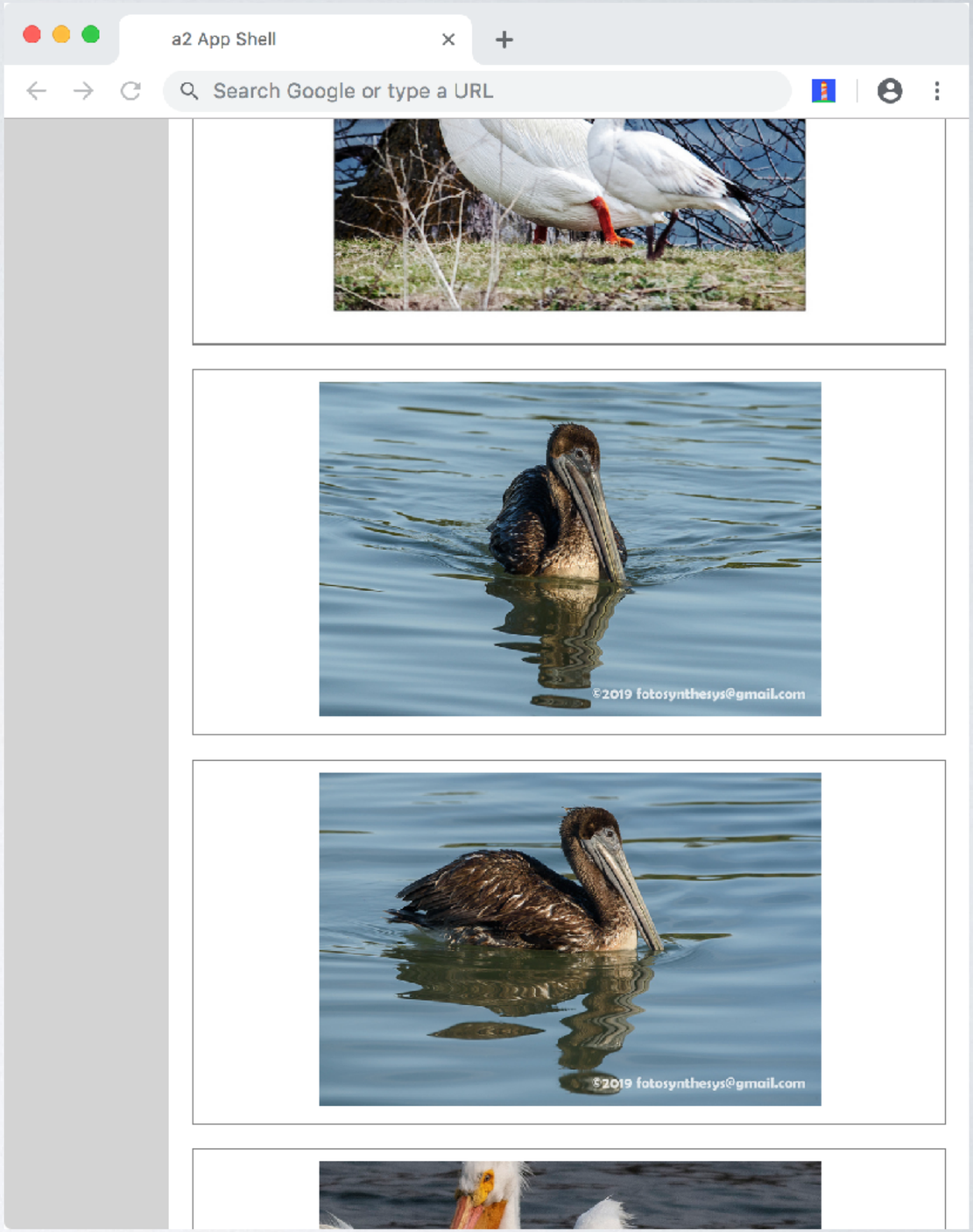
```
var x = new ImageLoader(imageName);
```

You should also be able to pass the title of the image.

```
var x = new ImageLoader(imageName, imageTitle);
```

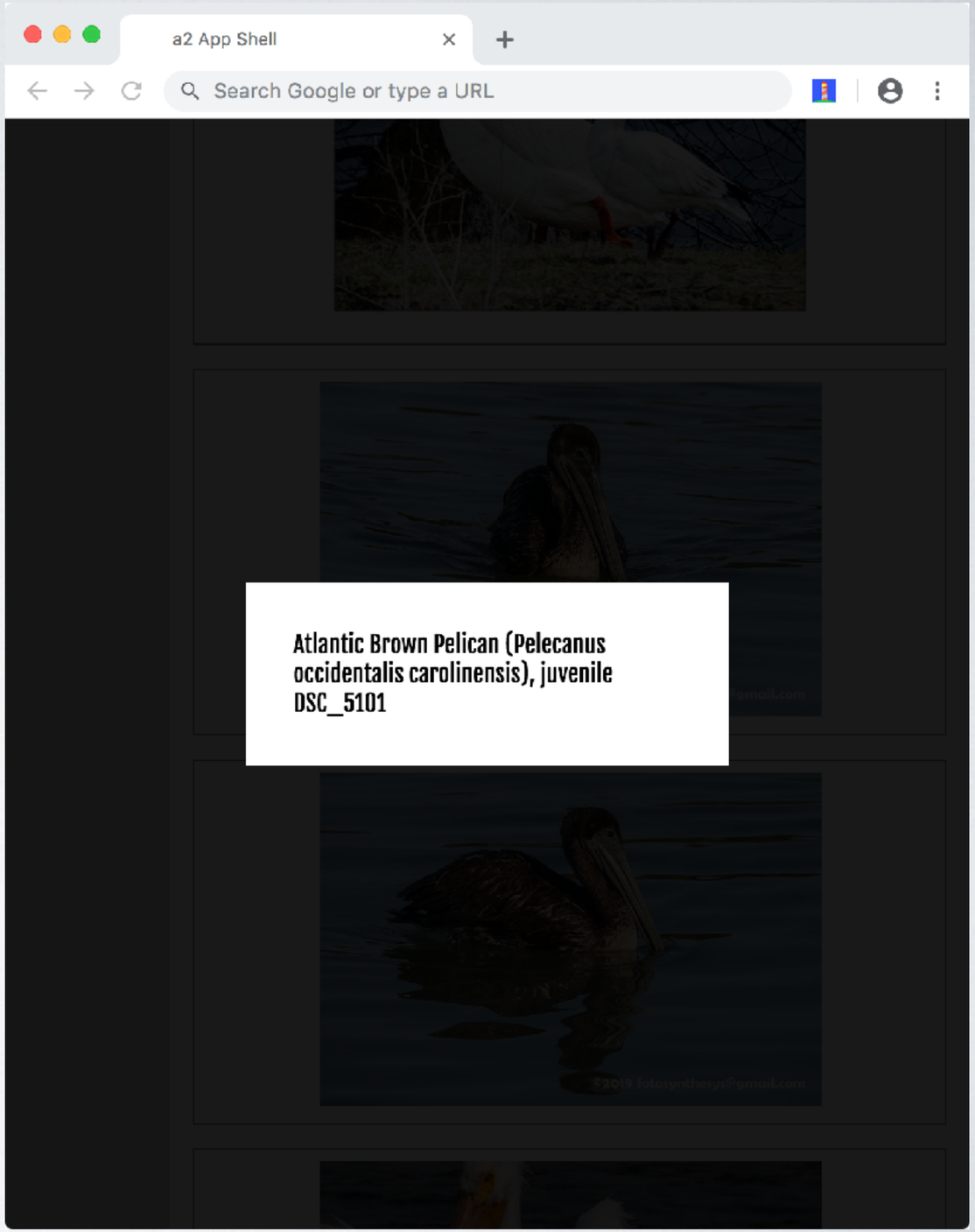
Then you can write a method that returns a Promise since it will be asynchronous. It should create an Image (DOM) Object from that filename and - once it finishes downloading - appends it to the page (and resolves the Promise).

Alternatively it could return that image object in the Promise (i.e. resolve with the Image object) when it finishes downloading.



Each image object should have an **onclick** event handler as discussed previously, i.e. clicking on the image displays the title. Therefore, it will need access to the image's title.

Closures may help here.



Part 3: Search Scripts

This part involves searching a large JSON object (with 9254 properties) for specific data.

It is stored in a .js file using the JSON-P format.

This file must not be cached.

We will use an arbitrary data file for the purposes of this assignment.

In this case a large file of films and where to find their scripts.

https://github.com/matthewfdaniels/scripts/

https://github.com/matthewfdaniels/scripts/blob/graphs/movieObj.js

```
processFilms({ '1':
   { scrape_id: '1',
     title: '10 things i hate about you',
     source: 'cornell',
     year: '1999',
     link: 'http://www.dailyscript.com/scripts/10Things.html',
     imdb_match: 'tt0147800',
     cornellId: 'm0' },
  '2':
   { scrape_id: '2',
     title: '1492: conquest of paradise',
     source: 'cornell',
     year: '1992',
     link: 'http://www.hundland.org/scripts/1492-ConquestOfParadise.txt',
     imdb_match: 'tt0103594',
     cornellId: 'm1' },
  '3':
   { scrape_id: '3',
     title: '15 minutes',
     source: 'cornell',
     year: '2001',
     link: 'http://www.dailyscript.com/scripts/15minutes.html',
     imdb_match: 'tt0179626',
     cornellId: 'm2' },
  '4':
   { scrape_id: '4',
     title: '2001: a space odyssey',
     source: 'cornell',
     year: '1968',
     link: 'http://www.scifiscripts.com/scripts/2001.txt',
```

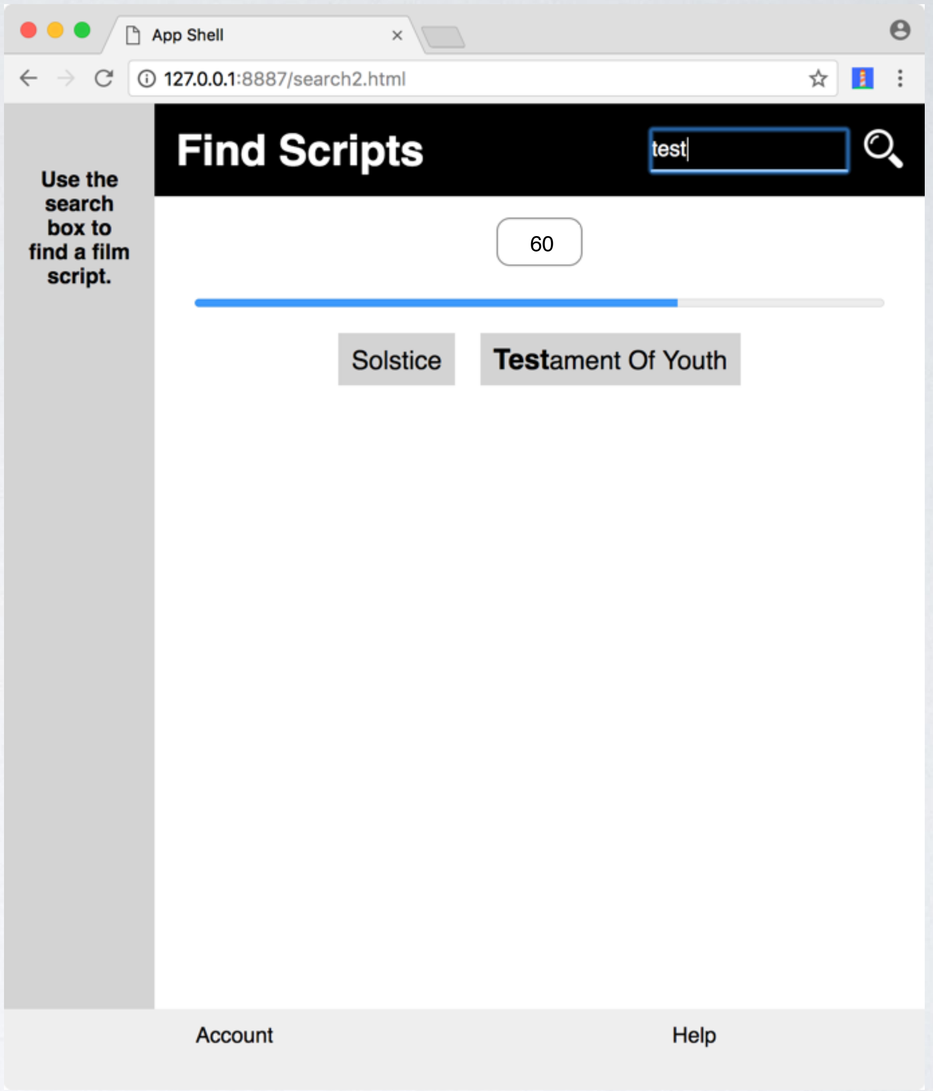
This data will be provided for you.

It was adapted slightly from the original to support **JSON-P**

```
processFilms( '1':
   { scrape_id: '1',
     title: '10 things i hate about you',
     source: 'cornell',
     year: '1999',
     link: 'http://www.dailyscript.com/scripts/10Things.html',
     imdb_match: 'tt0147800',
     cornellId: 'm0' },
  '2':
   { scrape_id: '2',
     title: '1492: conquest of paradise',
     source: 'cornell',
     year: '1992',
     link: 'http://www.hundland.org/scripts/1492-ConquestOfParadise.txt',
     imdb_match: 'tt0103594',
     cornellId: 'm1' },
  '3':
   { scrape_id: '3',
     title: '15 minutes',
     source: 'cornell',
     year: '2001',
     link: 'http://www.dailyscript.com/scripts/15minutes.html',
     imdb_match: 'tt0179626',
     cornellId: 'm2' },
  '4':
   { scrape_id: '4',
     title: '2001: a space odyssey',
     source: 'cornell',
     year: '1968',
     link: 'http://www.scifiscripts.com/scripts/2001.txt',
```

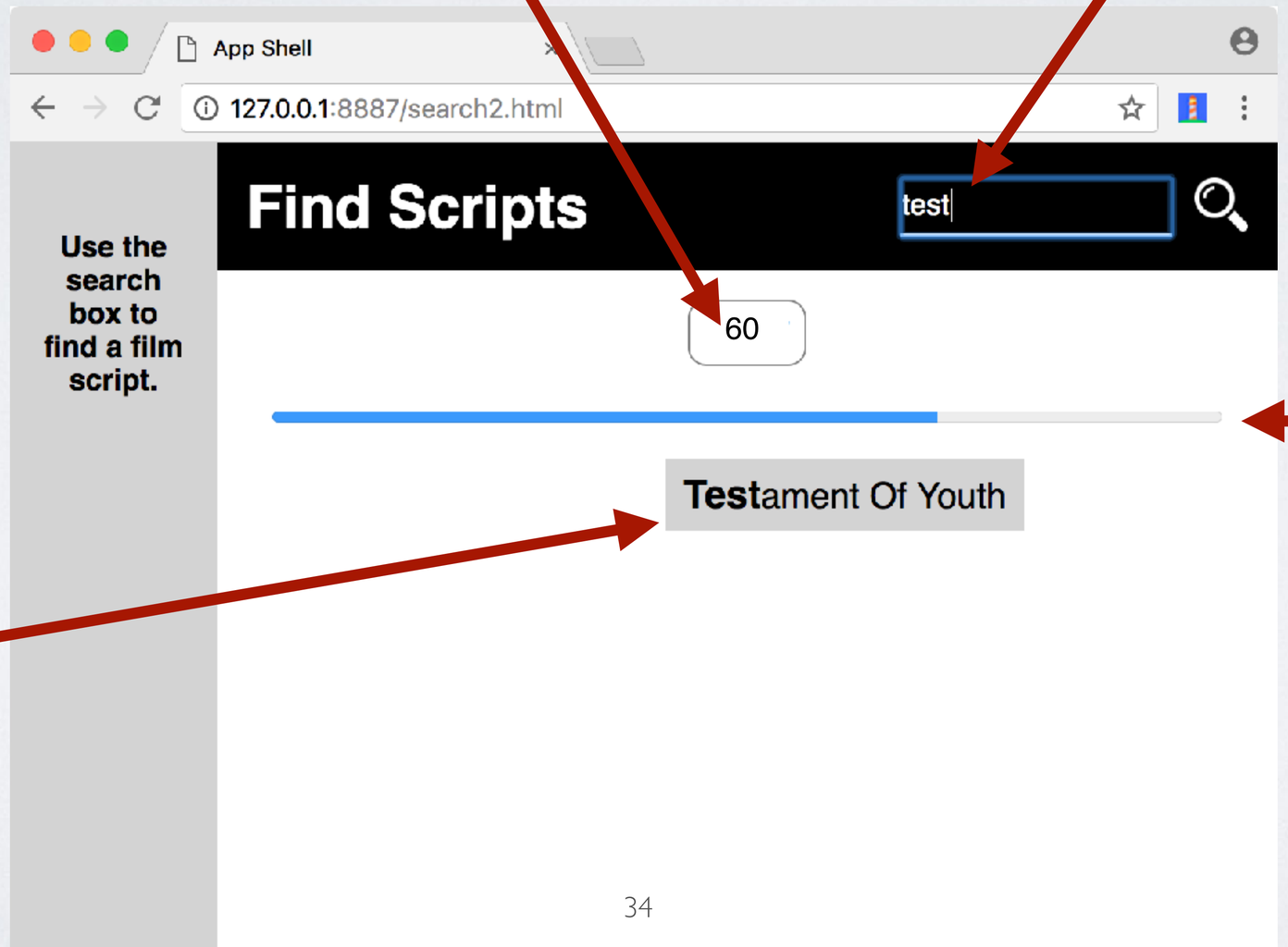
I.e. the function call to **processFilms()** was added.

You can adapt the app shell HTML/CSS from assignment I as before.



Percentage searched as number

Search field



Progress Bar (showing percentage searched)

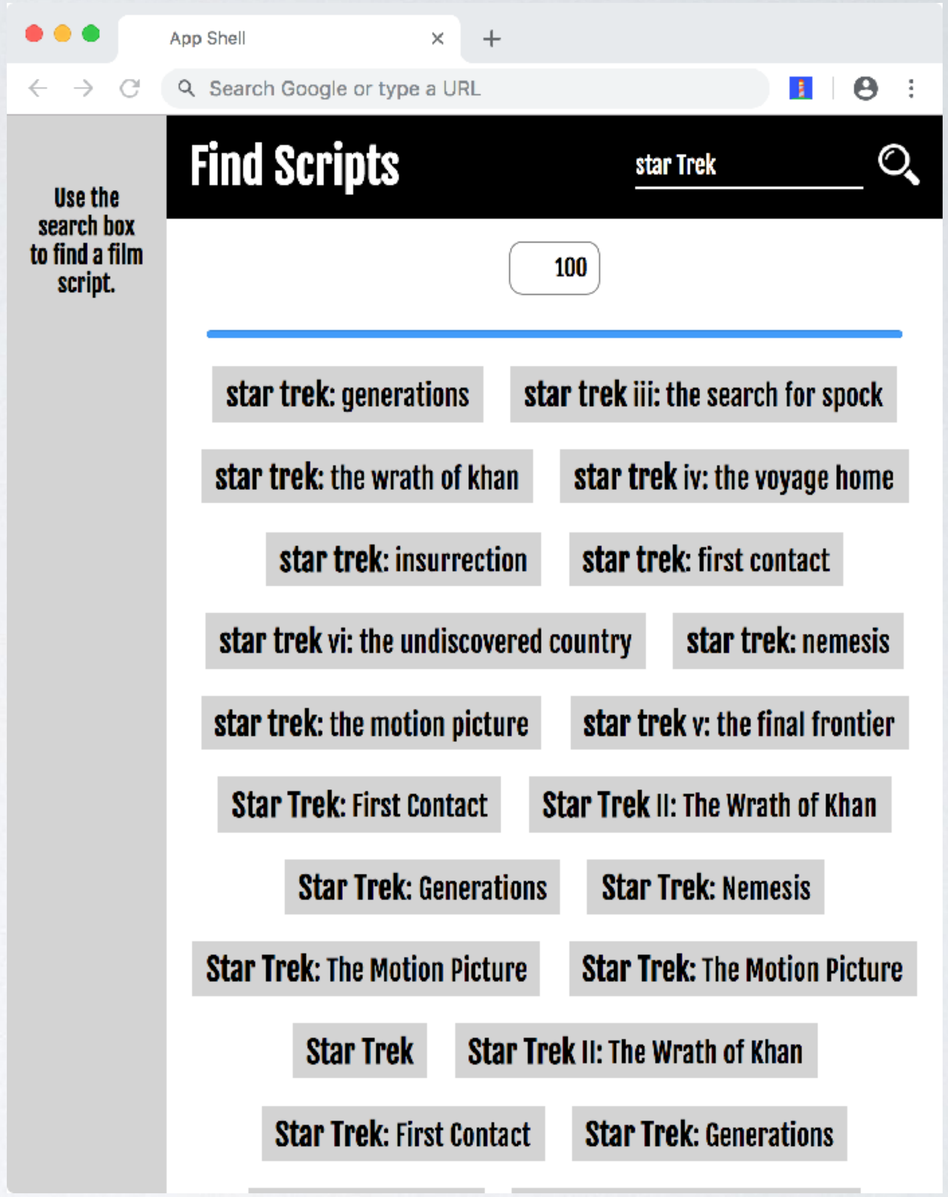
Results

The results are added to the page via DOM scripting as buttons (or similar elements).

If you click on a button it will open a new window/tab with that script.

```
window.open(url);
```

Note, not all urls in the dataset are still live. That is not an issue for this assignment.

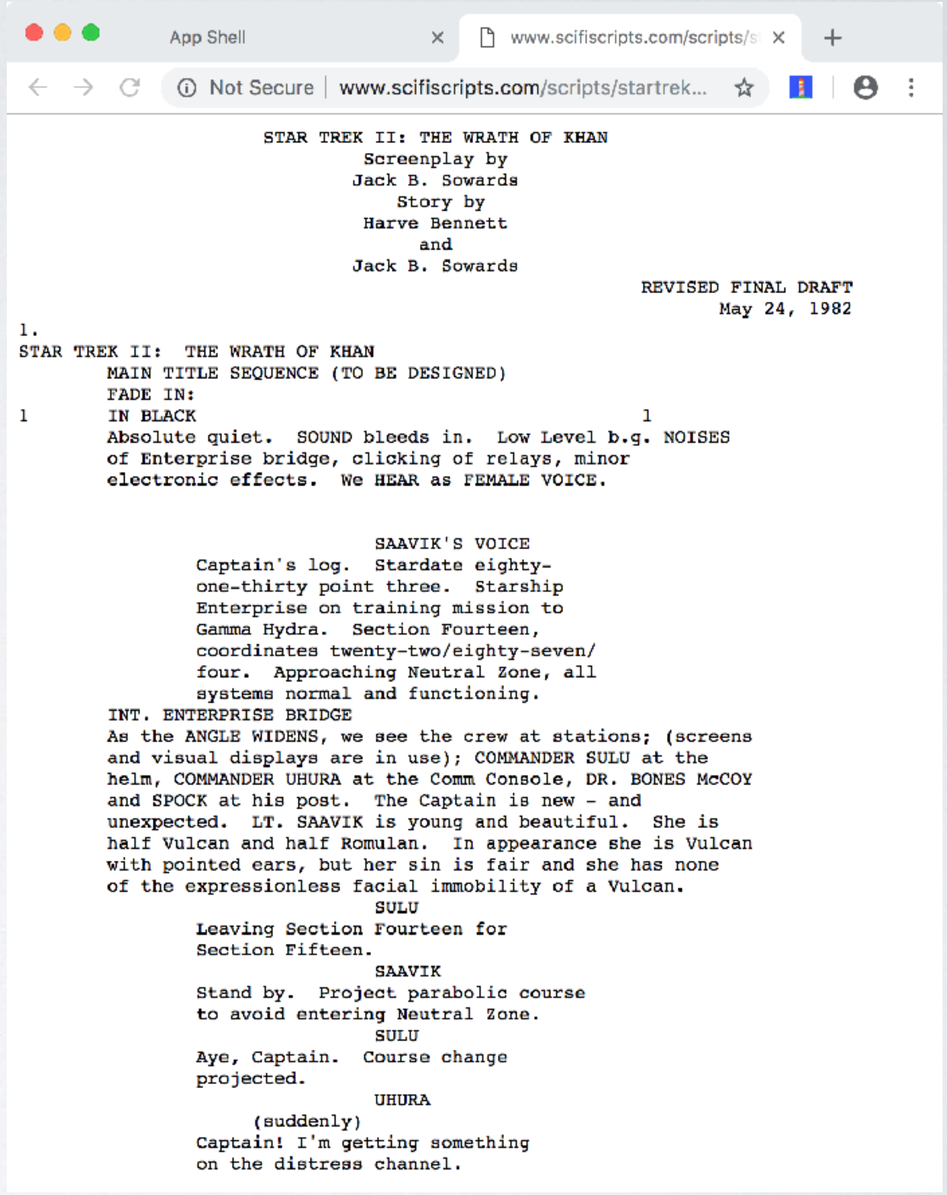


The results are added to the page via DOM scripting as buttons (or similar elements).

If you click on a button it will open a new window/tab with that script.

```
window.open(url);
```

Note, not all urls in the dataset are still live. That is not an issue for this assignment.



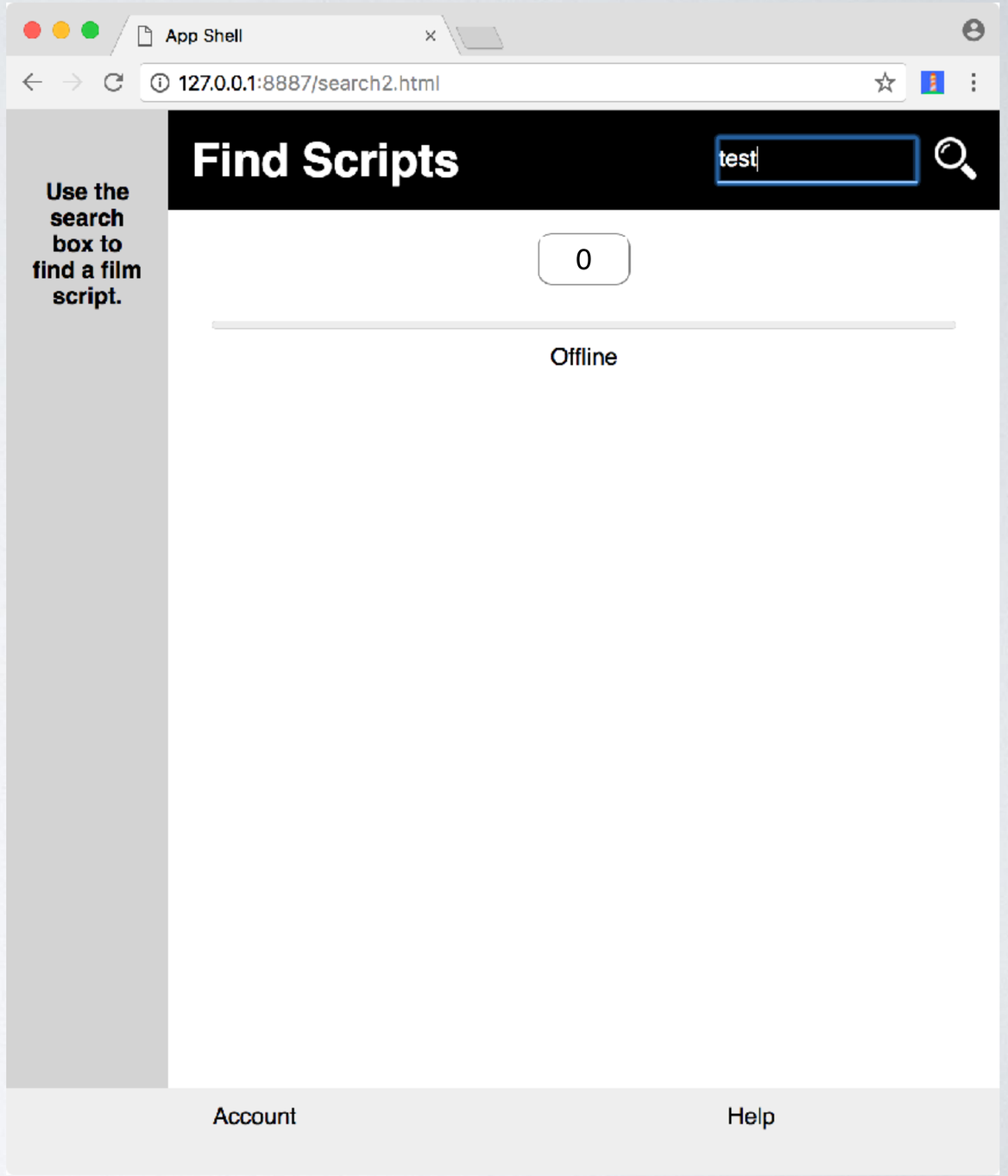
When **searching the data** you should check if the search term is contained in the title of a film (looping through all 1000 films).

If you find a match you should arrange for the **title** and **url** to be added to the page.

The page should still appear when offline.

However if you try a search it should inform you that it is offline instead of showing results.

NOTE: we could have cached the JSON-P file but we are emulating a web service so **we will deliberately avoid using a cached version**.



As you perform the search (and dynamically add the result buttons), you show what percent of the movies you have searched in the progress bar.

You also show the percentage as text on the page.

See included screencasts.

Since the constantly changing timer will be making a lot of use of the single thread in your page (to update the UI) you will search the file in a separate thread via web workers.

The web worker should do the following in response to a search term sent to it:

Automatically **search though each film** (i.e. loop through the array) in the JSON object

For each film, **check the title** for occurrences of the search term

If you find a match **send the (altered) title and the link** for that film back to your main script (see next slide for how the title is altered).

As your search continues it should **update your main script on its progress** so it can update the progress bar and percentage completed text field. I.e. you should send it the percentage of the films you have searched. .

You should also inform the main script **when the search is finished**.

You should convert the text that contained the match to highlight the match with a <mark> tag **before you send it to the main script**.

E.g. Assuming a search for "Night" you would convert the following:

```
night of the living dead
```

to:

```
<mark>night</mark> of
the living dead
```

Development Tips

The processing of nearly 1000 objects may take time when testing your code.

To help with your development you might find it useful to **arrange to only return a maximum of 100 results while testing** (so you don't have to wait for the entire search to finish everytime)

It may also help to speed up your work flow to **use a file with far fewer objects** when testing your code at the beginning.

In the screencasts provided the code returns all the matches it finds.

If you search for simple terms like "a" you will get 1000s of results.

Due to the length of the searches, this makes it easier to see the progress bar functioning when testing it with more generic search terms.

Reduce Number of Messages

Communication between Web Workers and your main thread can slow your code.

Assuming you are sending percentage values to the main thread so it can draw the progress bar, the Web Worker could ensure the values are unique before sending them. I.e. theres no point in sending a value like 5% a 100 times. Instead we send it once, and don't send the percentage value again until it has changed to 6%.

JSON-P in Webworkers

Your webworker can use **importScripts()** to download the JSON-P file which will in turn call the **processFilms()** function when it finishes downloading.

```
onmessage = function(e){

    searchTerm = e.data;

    importScripts("movieObj.js");

}



function processFilms(data)
{

        < Perform Search>

}
```

Using regular expressions to wrap text around a given match.

```
var regex = new RegExp("word",'ig');

var text = "There is a word here";

newText = text.replace(regex , '<mark>$&</mark>');

console.log(newText);
```

text we are looking for.

Options string:
i: case insensitive
g: global (i.e. match every occurrence, not just the first)

```
var regex = new RegExp("word",'ig');
```

text we will be searching

```
var text = "There is a word here";
```

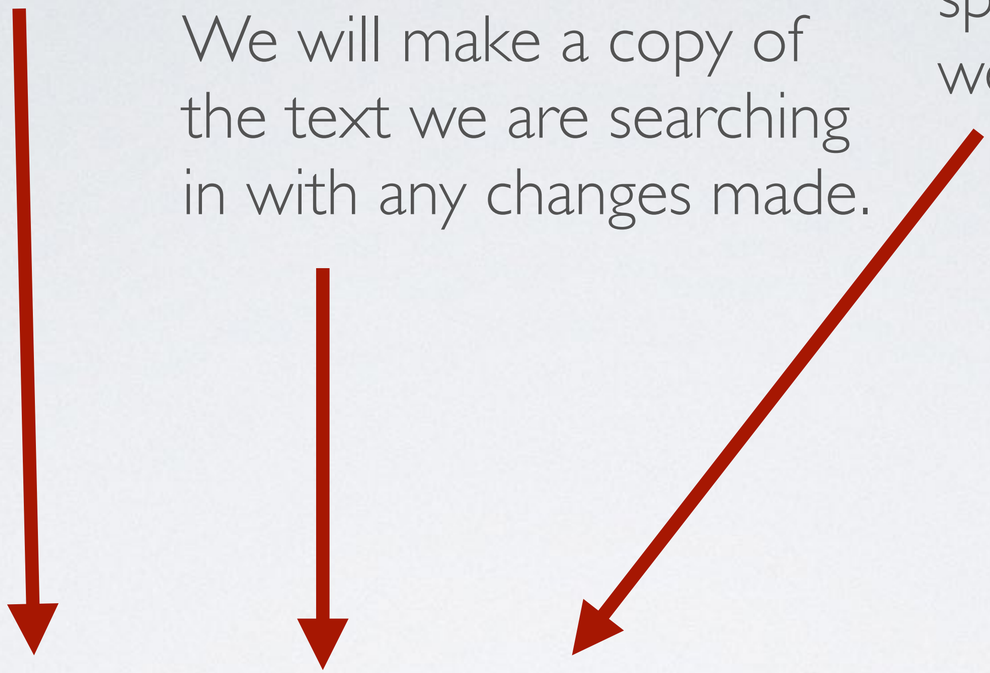
```
newText = text.replace(regex , '<mark>$&</mark>');
```

```
console.log(newText);
```

text we are searching in

We will make a copy of the text we are searching in with any changes made.

Regular expression specifying the match we are looking for.



```
newText = text.replace(regex , '<mark>$&</mark>');


console.log(newText);
```

This is what we will replace the matched text with.

We want to replace the matched text with the same text but with <mark> and </mark> tags around it.

$& will contain the matched text ("word" as per the regular expression)

```
newText = text.replace(regex , '<mark>$&</mark>');


console.log(newText);
```

```
var regex = new RegExp("word",'ig');

var text = "There is a word here";

newText = text.replace(regex , '<mark>$&</mark>');

console.log(newText);
```

There is a <mark>word</mark> here

Communication

Note that the communication from the WebWorker back to the main thread should be via JSON objects. This way we can send titles and descriptions with one message.

Part 4: Make your PWA Installable as app

Arrange for your PWA to be installable on devices.

I.e. It should have:

a responsive icon
an official app name
a splash/loading screen

It should display without browser chrome.

Note: PWA manifests don't set icons in somew versions of iOS

You can use the older **apple-touch-icon** link instead (you will need several of these).

```
<link rel="apple-touch-icon" sizes="512x512" href="polaroid-512.png">
```

You can find sample icons (for the purposes of this assignment) on sites like:

iconfinder.com

Each icon is usually available in different sizes.

Part 5: Caching Policies

You will need to write a service worker (shared by the two files) that will manage your caching policy.

The next pages contain a description of the caching policy that you must implement.

The **Caching Policy** is essentially a description of what the Service Worker will do in response to a request from one of your pages.

It describes where it will look for the response (the cache, the network, or generate the response itself), as well as the priority it gives to each source (e.g. search the cache first, if its not there check the network).

It also specifies when items are added to the cache (and whether some items are to be cached at all).

Different types of resources will require a different policy.

I.e.

App Shell Files (HTML, JavaScript, CSS and images, etc)

Flickr Search Result (i.e. JSON-P from flickr.com)

movieobj.js (JSON-P file containing script data)

Images from Flickr

The Policy

App Shell Files (HTML, JavaScript, CSS and images, etc)

Basic files needed for your app shell to function should be immediately cached when the service worker installs.

HTML, CSS, images, .js, fonts, etc.

Cache Policy: Cache on installation / for every consequent request you should check the cache first, then the network if it wasn't in the cache (if found online then add it to the cache).

Flickr Search Results (i.e. JSON-P from flickr.com)

Don't cache the Flickr Search results (i.e. the **JSON-P** response, we will be caching the actual images)

Cache Policy: Check the Network first, if it can't be accessed then just send back fallback content indicating the app is offline (i.e. some JSON describing what happened).

movieobj.js (JSON-P file containing script data)

Don't cache the movieObj.js JSON-P file (i.e. film script data)

Cache Policy: Check Network first, if it can't be accessed then send back fallback content indicating app is offline.

Images from Flickr

Cache the Flickr images

Cache Policy: Check in the cache first, if they are not in the cache then check the Network (and if found online add it to the cache)

Occasionally if you can't find a file in the cache or online you want to send back data your script will understand.

In this assignment we have cases where a JSON/JSON-P resource is being requested. If not found you can easily generate a JSON response in the ServiceWorker* and send it back to the requesting page. This JSON should contain data that the page can use to detect something went wrong (e.g. it was offline).

(*ServiceWorkers also let you send back HTML pages, alternative images, etc.)

E.g. here we return our own JSON-P file instead of the one we failed to read online.

```
return responseFromFetch.then (

        function() {    <... return the resource ... >    }

).catch(

        function() {    return  new Response(

                          "showImages({offline: true})",

                          {headers: {"Content-Type": "text/javascript"}}
                        );
                    }
);
```

The response you send back should contain the JSON and set its **content-type** to **text/javascript**.

JSON-P response

Set the header

```
return  new Response(

  "showImages({offline: true})",

 {headers: {"Content-Type": "text/javascript"}}
);
```

Testing & Development

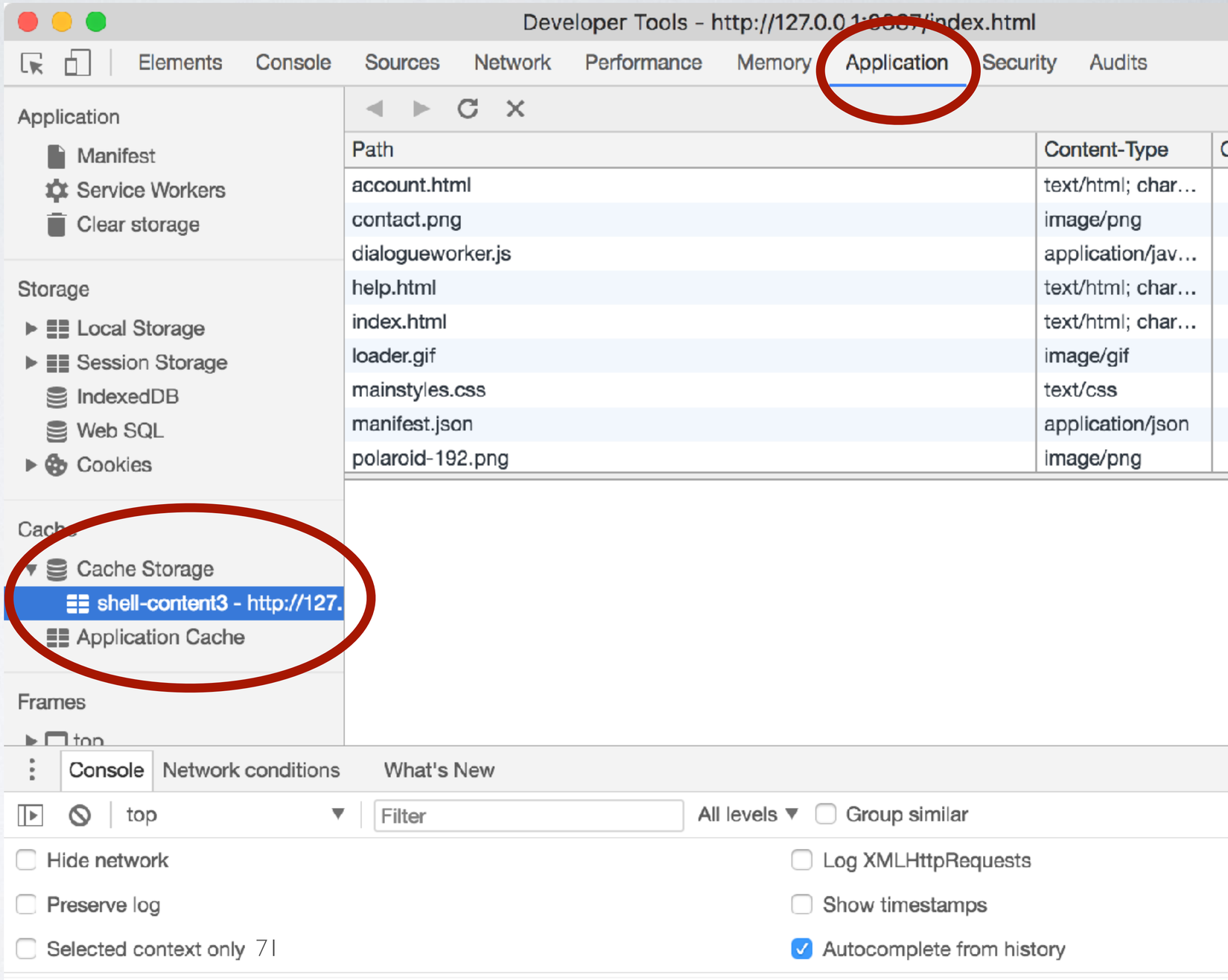
You can use tools to help develop your PWA.

For example, Chrome has features that allow you simulate various network conditions.

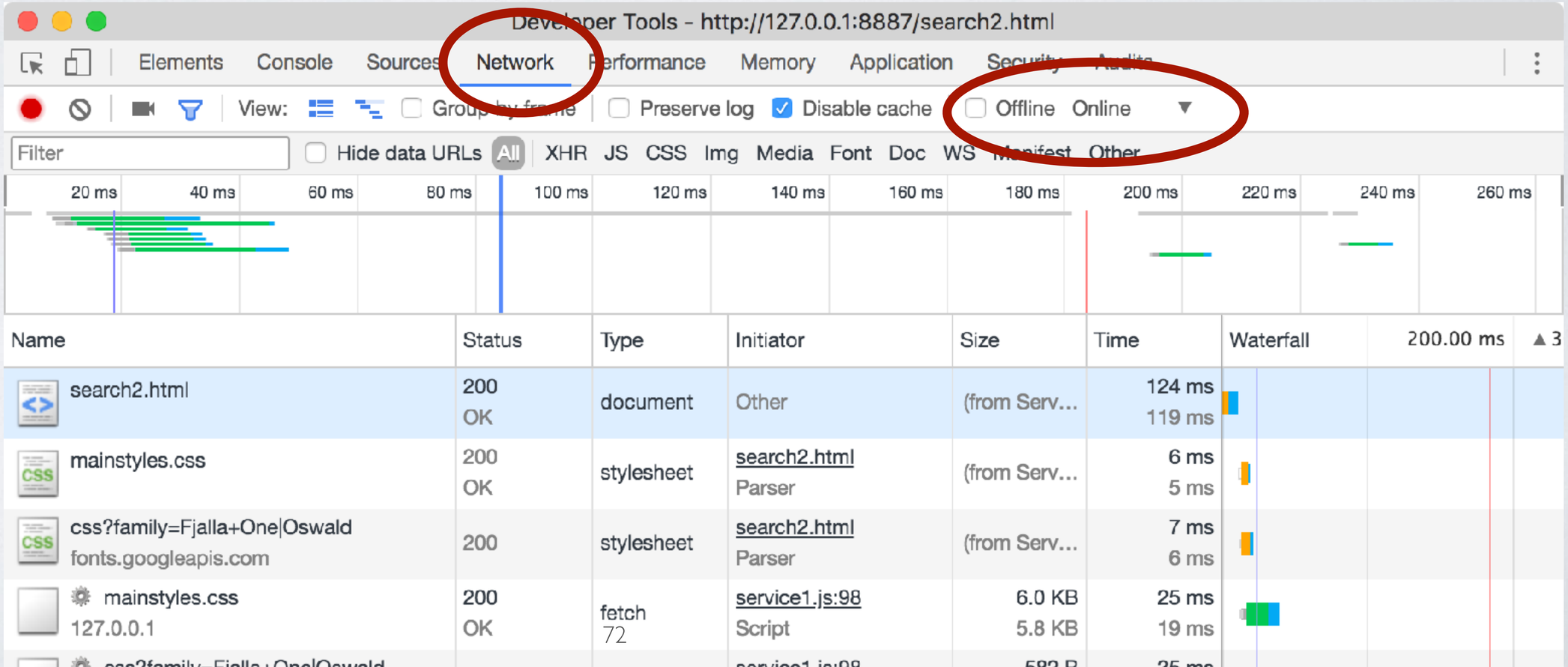
You will find them among the regular developer tools.

(Other browsers have similar tools.)

Here you can examine your cache (and delete the entire cache or individual files within it which is useful when you want to test the initialisation phase multiple times).

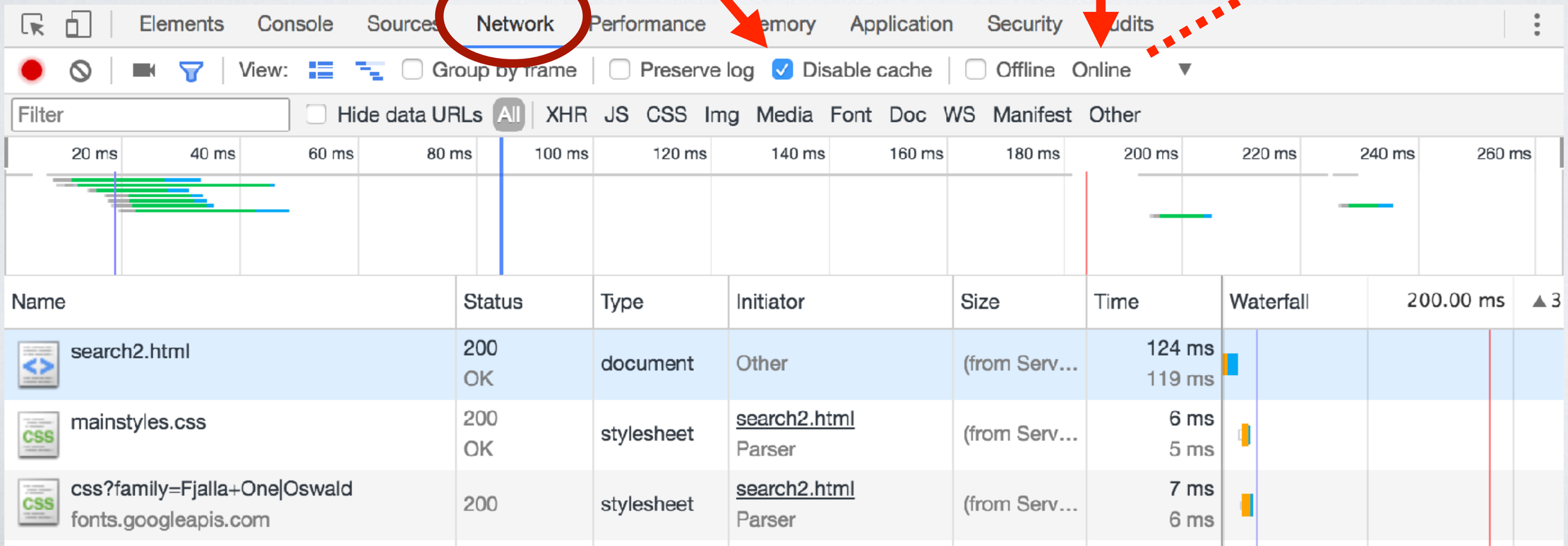


You can emulate being offline.

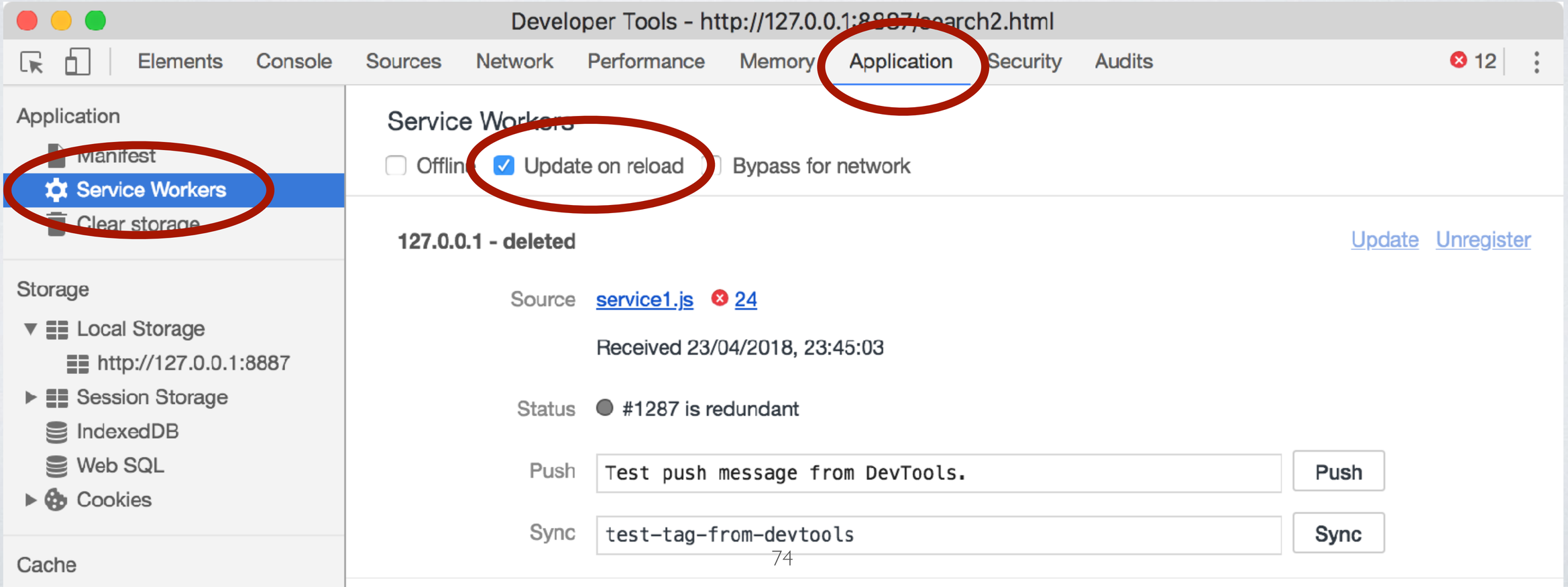


While developing your app you should disable the cache (this is the regular browser cache and not the cache used by your Service worker)

You can emulate slow network speeds to better see how you app operates. You can set this to *Slow 3G* or equivalent.



Usually you have to refresh a page two or more times for a change in a service worker to take effect (by design). Chrome provides a way to get immediate updates when testing your code.



Some features of PWAs require HTTPS. (Frequently this requirement is waived when you use **localhost** as your server for development purposes).

However, you will be provided secure hosting for your PWA.

Make sure you use https when accessing your pages.

i.e.

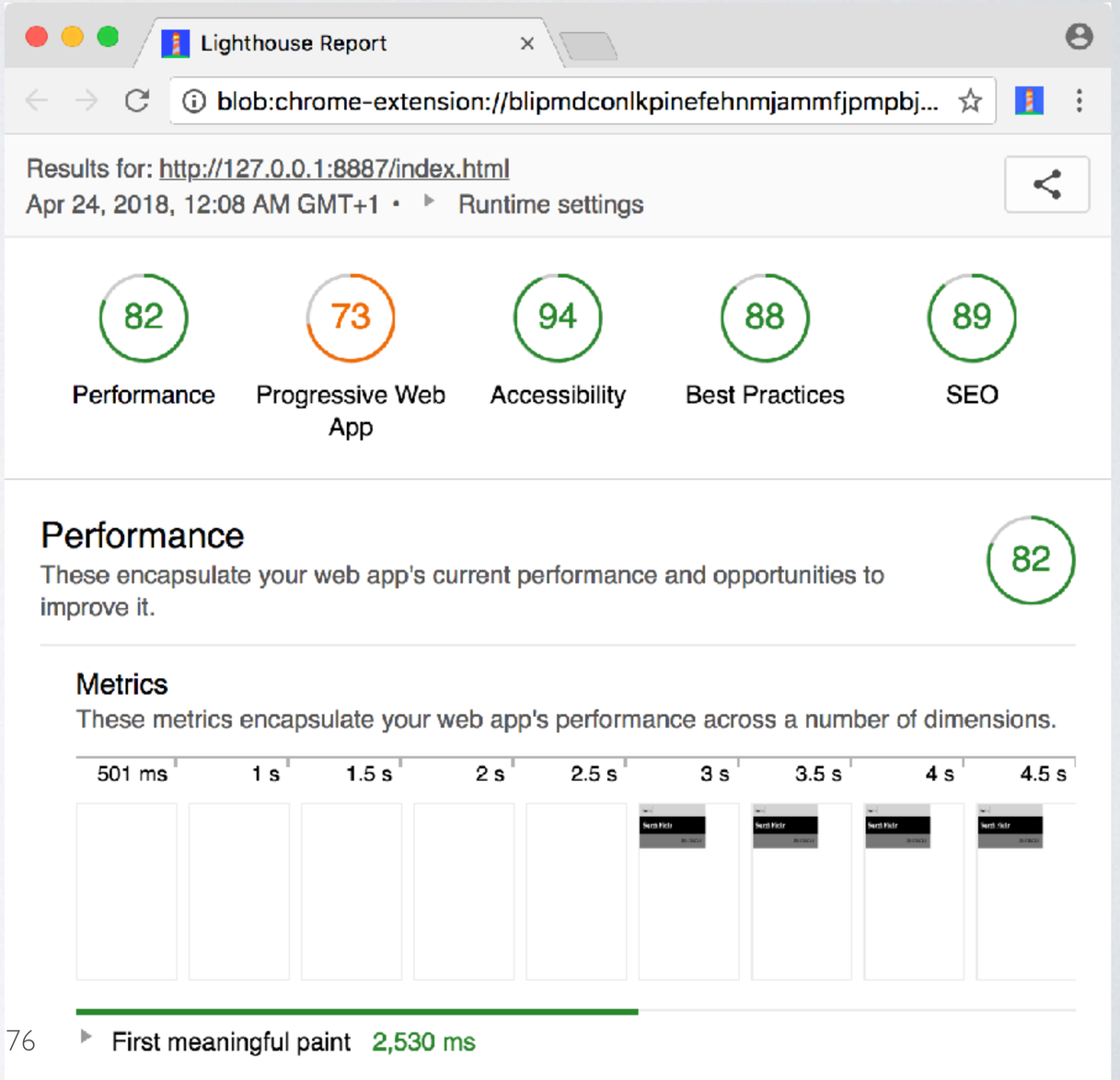
`https://webdevcit.com/.......`

not

`http://webdevcit.com/.......`

https://chrome.google.com/webstore/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk?hl=en

Lighthouse can check the performance of you PWA for you (although not a requirement for this assignment).



Submission

Your app should be available online on the server provided for you.

Your final PWA should be installable from that server.

You must also submit the code in a zipped folder to **Canvas**.

Your main page should be called **index.html**

Make sure both pages are reachable from this page.

You should also email me a URL to your PWA on the server.
Your email should have the subject:

PWA 2020: Assigment 2 Submission

See also the **Assignment Regulations** document.

All code should be your own. You cannot use frameworks.