Progressive Web Apps

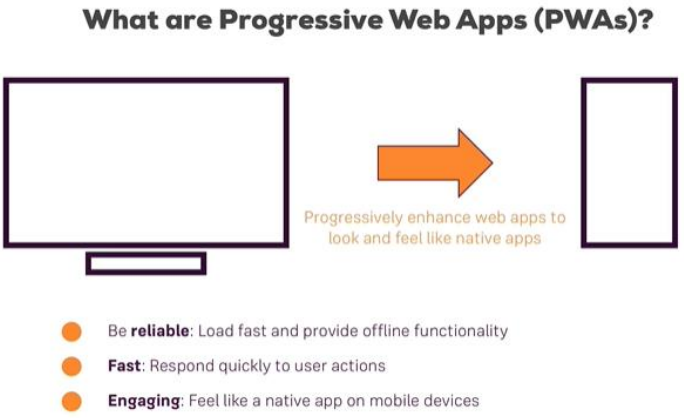
Sources

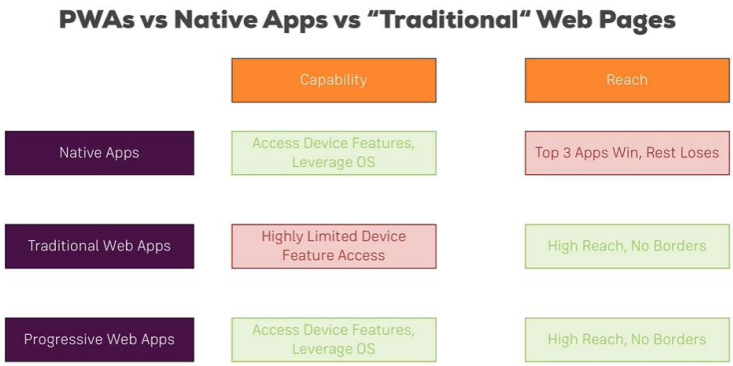
- [Udemy] Progressive Web Apps (PWA) - The Complete Guide (Link)

Section 1 – Getting Started

General Information

- Progressive web apps (PWA) are a set of features and technologies that can be added to existing web applications to turn them into native app-like or native mobile-like applications.
    - Features like home screen install, offline accessibility, push notifications, user location, device camera access, and background sync are all part of the PWA experience.
- The idea is to progressively enhance web apps to look and feel like native mobile applications.
    - As well, the web application should still work on older devices or browsers, although it would not be quite the same level of experience.
- Note that Chrome Developer Tools is an excellent tool for working with web pages and apps.
    - We can simulate both device screen sizes and offline modes with the tools, which can help us see what PWA offline capabilities look like, for instance.
    - Chrome also has a feature under the 'Audits' section that runs lighthouse testing on a PWA site to determine how well it is implemented.
    - We should make sure that 'disable caching' is checked in the network tab.





Setup

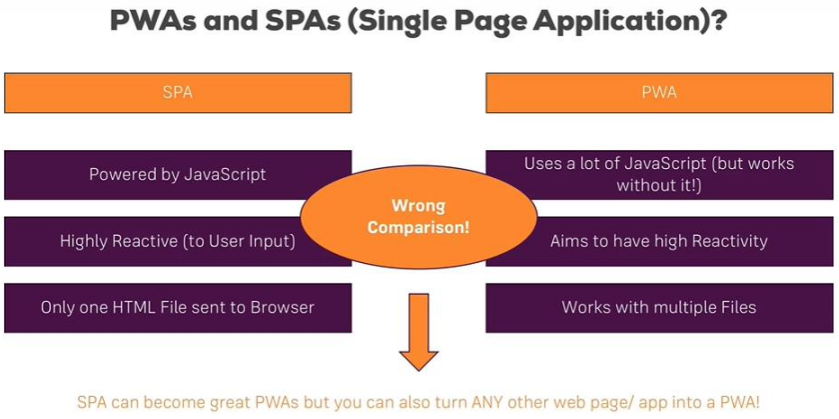
- Just like for Angular Apps, we setup new PWA projects by running npm install (or npm install --save http-server@0.9 --save-exact) in the project folder that we will be working in. This will install all the dependencies that the project requires.
    - We need to have the http-server dependency installed so that a server is running http. Note that if we just double-click on the html files, we will be using file protocol, which is not http.
    - We then run the server with npm start and keep the terminal window open so that the terminal keeps running.

Core Building Blocks

- Service workers are the core building block for PWAs and are basically JavaScript files that run in the background, even when the application is closed.
    - They also allow for background sync, caching and offline support, and push notifications.
- The application manifest is another key building block since it allows the PWA to be installed and added to the home screen.
- Although he will not really cover it in this course, responsive design is a key building block for PWAs.

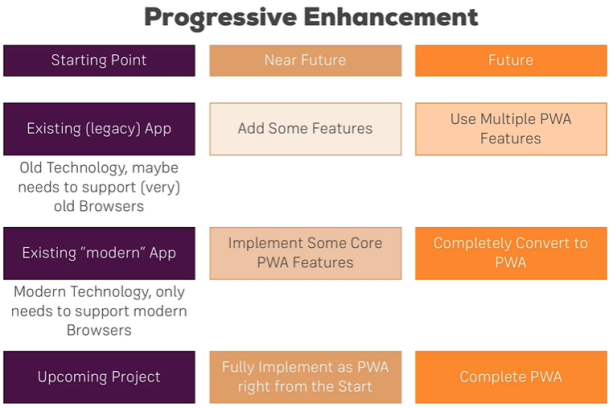
Single Page Applications

- Single page applications (SPA) are applications powered by JavaScript and built with React, Angular, or View that renders a website in a single page rather than multiple web pages.
    - PWAs are totally separate from SPAs, and any SPA can be made into a PWA, same as any multi-page web application could also be made into a PWA.
    - We want PWAs to load fast compared to even SPAs.



Progressive Enhancement

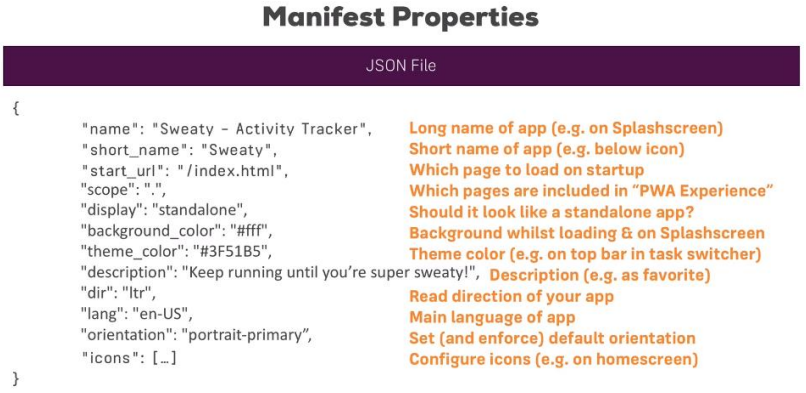
- We want to be able to progressively enhance our application.
    - This means that we want to be able to add only those enhancements that make sense for our application.
- Older web applications can absolutely be enhanced with modern features to make it into a PWA or leave it at some state in between.

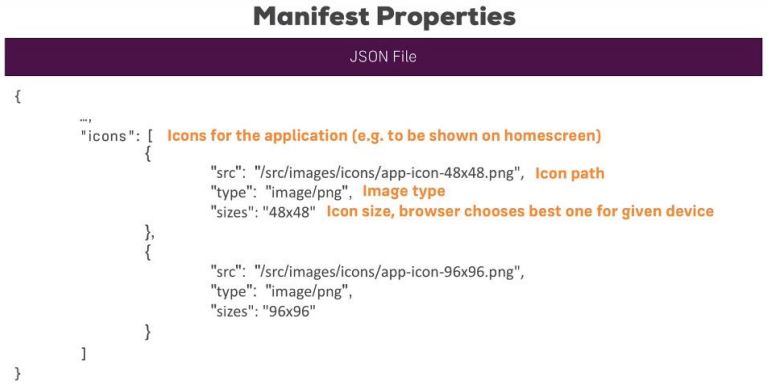


Section 2 – Understanding the App Manifest

General Information

- The web app manifest (manifest.json) is a single file that we add to our project that allows us to pass some extra information to the browser.
    - With certain browsers, the web app manifest can even permit the application to be installed to the home screen of the browser or operating system, allowing it to look and feel like a native app.
- To use the manifest, the manifest must be created and then added into the <head> of each webpage. On a SPA, this would only have to be added once, perhaps to the index.html.
- A list of icons is included in the manifest, and the browser will then choose the best icon for the given device.
    - The icon will be displayed on the splash screen as well.
- Browsers that do not support the web app manifest will not fail when loading the page but will simply ignore the manifest file.
- To view the web app manifest details in Chrome Developer Tools, choose the Application tab and select Manifest.





Running App on Android Emulator

- To run the device on an Android Emulator, Android Studio should first be installed.

- Go to the Android Virtual Device (AVD) screen, and create a new Google Play enabled virtual device.
- With the application running in the IDE (npm start), browse to 10.0.2.2:8080 in the Chrome browser in the AVD.
- The PWA can then be added by choosing to Add to Home Screen in the drop-down menu.

- Note that I had problems with the AVD since I could not update from an older version of Google Chrome without it breaking and not loading webpages.

Running on Safari and Internet Explorer

- We can add a few metadata items to the index.html file in order to have additional functionality with Safari.

```
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="black">
<meta name="apple-mobile-web-app-title" content="PWAGram">
<link rel="apple-touch-icon" href="/src/images/icons/apple-icon-57x57.png" sizes="57x57">
```

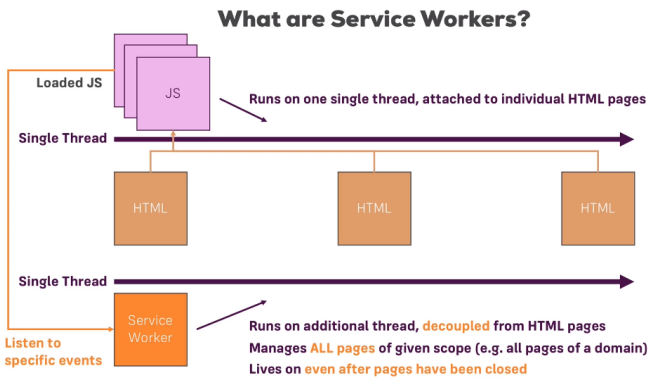
- Metadata tags can also be added for internet explorer to allow for some PWA functionality.

```
<meta name="msapplication-TileImage" content="/src/images/icons/app-icon-144x144.png">
<meta name="msapplication-TileColor" content="#fff">
<meta name="theme-color" content="#3f51b5">
```

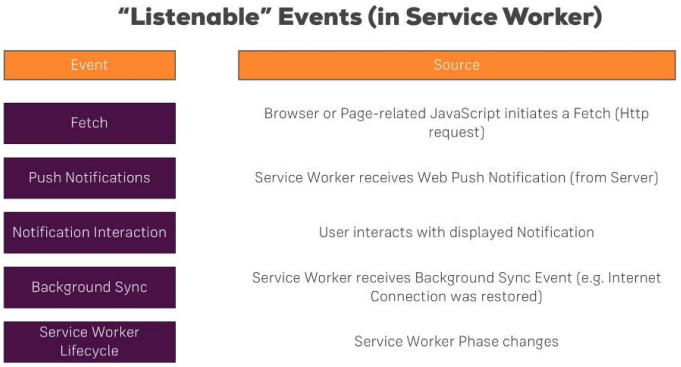
Section 3: The Service Workers

General Information

- Service workers allow us to make our web application offline capable, as well as allow for push notification and caching.
  - These service workers run on a separate thread than the main application and are decoupled from the HTML pages.
  - They are a background service that can run even after the browser is closed.
  - Service workers are great for listening and reacting to events.



- Service workers can listen to fetch events, which are http requests.
  - Every request sent by the browser to load some assets goes through the service worker.
  - Since most requests go through the service worker, we then have ways of manipulating these requests.
- They can also react to push notifications received from a web server and can then create a notification for the user on the device.
  - We can react to users interacting with the push notification in the service worker. It is important that the service worker capture these push notifications since the service worker is always running, versus the application JavaScript code that only runs when the application is open.

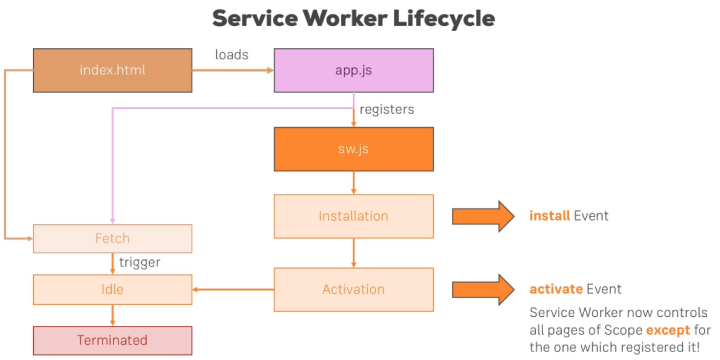


- Background synchronization can be used, which will allow a user interaction to be stored when there is weak network connection and executed once internet connection is re-established.
  - The browser will typically emit an event when connection is re-established, and the service worker can listen for and react to that event.
- Note that we can have multiple service workers for a website, but only with different scopes.
  - The more specific service worker overwrites any others for its scope.
- Web workers are different from service workers, although they do not live after the page is closed where service works can.

Service Worker Lifecycle

- The app.js file in the main application can register a service worker as a background service when run.

    - Service workers are installed once registered, but they may not be activated until the browser tab is reopened, depending on if an older version of the service worker already exists.
    - The scope of the service worker defines which parts of the page are controlled by the service worker.
- The registration code is always run every time that the application is opened, but a new service worker is not necessarily installed each time that the web application is visited.
    - The service worker is reinstalled only if the service worker code is changed by a single byte or more.
- After being idle for a certain amount of time, a service worker will terminate, which essentially means that it is sleeping.
    - However, the service worker will wake up when a new event comes in.



Registering a Service Worker

- Adding a `-c-1` flag to the start property inside of package.json file will ensure that we do not cache anything inside of the browser cache while developing the app.
    - For production, this flag is not required because we won't be changing our files as often.
- The scope of a service worker only applies to pages inside of the folder that it sits in.
    - Typically, the service worker is added in our root web folder (e.g. the public folder) so that it would apply to all files no matter whether they are in subfolders or not.
- While the service worker can be registered in HTML files, it is best to register it inside the app.js file for the application. It would then be available for all files.
    - We should first check to see if the browser that the user is using supports service works before registering the service worker.
    - The application ➔ service workers tab in Chrome Developer Tools shows information about the service worker one installed.
    - Note that the register method takes a scope property where the scope can be further restricted.

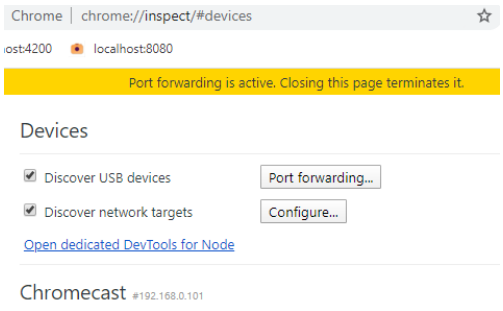
```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker
    .register('/sw.js')
    .then(function () {
      console.log('Service worker registered!');
    });
}
```

- The manifest.json file and the sw.js service worker are not related in any way, although the manifest file would be needed to install the service worker to the home screen.
- Service workers can only be deployed through https because of the amount of information that they intercept, so an encrypted connection is necessary.

Reacting to Events

- Service workers are made to handle events, so event listeners are included in the service worker.
    - Although we use the addEventListener(...) method, we will not be able to listen to DOM events.
    - We can now listen to events such as the service worker installation, activation, or any fetch events.
- It is important to note that while service worker installation may occur upon loading the PWA, the activation of the service worker might not take place until the tab is closed and the application is re-entered so that the page is not broken in the meantime.
- The fetch listener will only trigger when the web application fetches something.
    - We can use the respondWith(...) method in combination with the fetch listener to make our service worker a network proxy, where any requests going in and out will go through the service worker.

Remote Debugging with Device



- With USB Debugging enabled on your phone, plug the phone into the development computer by USB.
    - Browse to chrome://inpect#devices to see a list of devices connected to the computer. Enable port forwarding with port: 8080 and localhost:8080.
    - Now we should be able to access the development site with our android phone.
    - By clicking inspect, we can interact with the app on the device from Chrome Developer Tools.

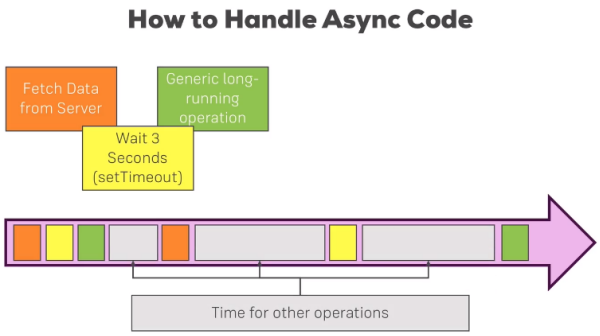
App Install Banner

```
function openCreatePostModal() {
  createPostArea.style.display = 'block';
  if (deferredPrompt) {
    deferredPrompt.prompt();

    deferredPrompt.userChoice.then(function(choiceResult) {
      console.log(choiceResult.outcome);

      if (choiceResult.outcome === 'dismissed') {
        console.log('User cancelled installation');
      } else {
        console.log('User added to home screen');
      }
    });

    deferredPrompt = null;
  }
}
```

- We can set when the app install banner shows in our app by creating an event listener for a button click or some similar event, and then show the app install banner once that event has taken place.
    - There is only one chance for this install banner to show to the user apparently, so if the user chooses to dismiss, it will not show up again.

Section 4: Promise and Fetch

General Information



- Although JavaScript is single-threaded, it can handle asynchronous code.
    - Callbacks are an example of asynchronous code, where a function is run after a specified amount of time has passed.

Promises

- A promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.
    - The promise object takes one argument, which is a function that will execute right away.
    - That function will also take two other functions are arguments – a resolve and a reject function.
    - Promises will always give us back a response of either resolve or reject.

```
promise.then(function(text) {
  return text;
}).then(function(newText) {
  console.log(newText);
}).catch(function(err) {
  console.log(err.code, err.message);
});
```

- We can then use a then(...) function that is run when the function completes, and multiple then(...) functions can be chained together.
    - An function to catch errors from reject can also be added to the then(...) chain to handle that type of return.
    - Keep in mind that the catch function will handle any errors in the then(...) chain that occurs prior to the catch function.
    - An example of a built-in promise being used is when register is called on a service worker in the app.js file. We setup a then() function to notify the console when the service worker was asynchronously registered.
- It is somewhat rare that we will create our own promises, and typically we just use the built-in promises.

Fetch

```
fetch('https://httpbin.org/ip')
  .then(function(response) {
    console.log(response);
    return response.json();
  })
  .then(function(data) {
    console.log(data);
  })
  .catch(function(err) {
    console.log(err);
  });
```

- The Fetch API provides an interface for fetching resources (including across the network).
    - Fetch provides a generic definition of Request and Response objects (and other things involved with network requests). This will allow them to be used wherever they are needed in the future, whether it's for service workers, Cache API, and other similar

things that handle or modify requests and responses, or any kind of use case that might require you to generate your own responses programmatically (that is, the use of computer program or personal programming instructions).

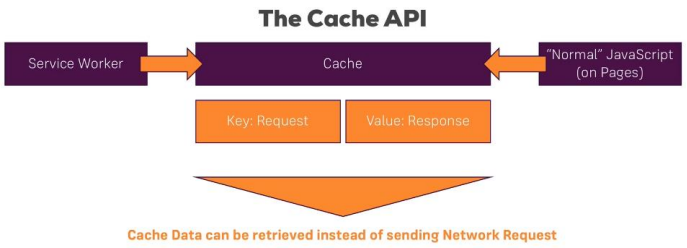
- Promise then(...) calls can be used in tandem with the fetch request, as shown in the above screenshot.
- Polyfills allow us to use fetch on older browsers.

Service Workers

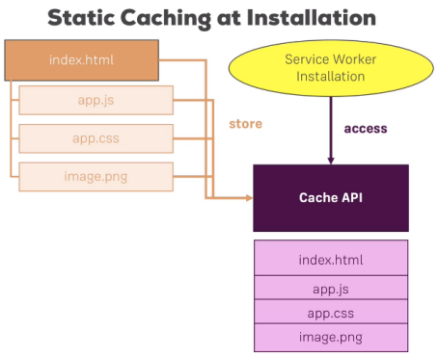
- We want to use fetch within our JavaScript files since that will allow us to intercept the fetch events in the service worker with the fetch listener.

Section 5: Service Worker – Caching

General Information



- The main function of service workers is to provide offline capabilities to the app.
- The browser has its own cache, but we can not rely on it since it is managed by the browser.
    - A separate cache, called the cache API, also exists and is one that we can manage.
- The cache API ("cache") holds data in a key-value pair format and stores a URL that is visited as the key and the contents as the value.
- The cache can be accessed by both service workers and the JavaScript for the website.
    - Using fetch will allow us to intercept any requests and can direct those requests to either the network endpoints or to the cache.
- The application shell is additional terminology we should know and consists of all the static content of our webpage (e.g. HTML, CSS, and JavaScript files).
    - The dynamic content is often desirable to cache, although it is not part of our app shell.



Static Caching

- During the service worker installation is when we would typically store the static pages into the cache.
    - We must be careful to not overpopulate the cache since the browser may clean it up if it runs out of space.
- We can create a new cache or open an existing one within the cache API by using the caches.open(...) function, which returns a promise.
    - Caches can be viewed by going to the Application ➔ Cache Storage in Developer Tools.
    - To ensure that the cache is created or opened before any further manipulations occur on the cache, we can use the event.waitUntil(...) method.
- Static resources are added to the cache by providing their paths from the root of the project.
    - Keep in mind that when we add an item to a cache, the item is stored as a key-value pair where the key is the path in the form of a request (in the case of static resources) and the value is the contents of the file.

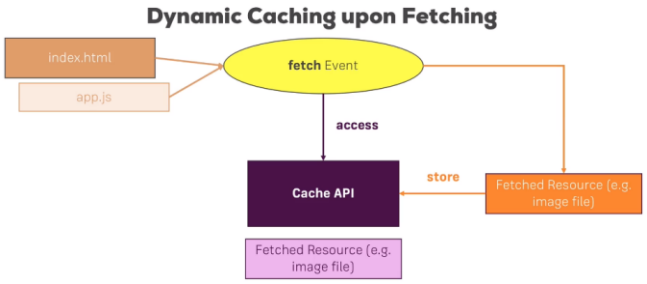
```
var CACHE_STATIC_NAME = 'static-v4';
var CACHE_DYNAMIC_NAME = 'dynamic-v2';

self.addEventListener('install', function(event) {
  console.log('[Service Worker] Installing Service Worker ...', event);
  event.waitUntil(
    caches.open(CACHE_STATIC_NAME)
      .then(function(cache) {
        console.log('[Service Worker] Precaching App Shell');
        cache.addAll([
          '/',
          '/index.html',
          '/src/js/app.js',
          '/src/js/feed.js',
          '/src/js/promise.js',
          '/src/js/fetch.js',
          '/src/js/material.min.js',
          '/src/css/app.css',
          '/src/css/feed.css',
          '/src/images/main-image.jpg',
          'https://fonts.googleapis.com/css?family=Roboto:400,700',
          'https://fonts.googleapis.com/icon?family=Material+Icons',
          'https://cdnjs.cloudflare.com/ajax/libs/material-design-lite/1.3.0/material.indigo-pink.min.css'
        ]);
      })
  )
});
```

- We can then configure our fetch event listener to determine whether the item is in the cache already and respond with that item if it is. If the item is not in the cache, then the item will be requested from the network.
    - If the item does not exist in the cache, the request response passed to the then(...) function will be null.
- The root of the application ('/') also needs to be cached, since that is the first page that loads.
- Keep in mind that even if we do not necessarily need files (e.g. polyfill files for older browsers), we should still cache them since it will improve performance on modern browsers by making it so they do not have to reach out to the network for the files.

```
self.addEventListener('fetch', function (event) {
  event.respondWith(
    caches.match(event.request)
      .then(function (response) {
        if (response) {
          return response;
        } else {
          return fetch(event.request)
            .then(function (res) {
              return caches.open(CACHE_DYNAMIC_NAME)
                .then(function (cache) {
                  cache.put(event.request.url, res.clone());
                  return res;
                })
            })
            .catch(function (err) {
            });
        }
      })
  );
});
```

Dynamic Caching

- We can add items dynamically to the cache, which is useful if a user visits a page that was not part of the static cache, for instance.
    - In our fetch event listener, we can capture any requests that are not stored in the static cache and put them in a dynamic cache instead. This is shown in the above screenshot.
    - Note that we must clone the response since the original response is consumed when it is returned, and hence a copy of the request is required.



Additional Notes

- Fetch errors can be caught by using the catch(...) method inside of the fetch event listener.
    - We do not want to cache our service worker because that would cause the application to enter an infinite loop where old versions of the service worker are fetched from the cache rather than grabbing a new version on the network.
- We may have a problem where json data (or similar) is being dynamically cached, even though the data frequently updates.
    - Ideally, we only want to be storing images, CSS, JavaScript, and HTML in the cache.
- One problem that we may run into is that if we change something in one of our CSS files, for instance, that change will not be reflected in the PWA until a change is made to the service worker, at which point an updated service worker will be installed and all files will be recached.
    - One way to get around that is to add version numbers to our caches (e.g. 'static-v2') inside of the service worker. The PWA would then see that the service worker was updated and would download a new version and rebuild the cache. This also ensures that the old cache is not corrupted in the meantime.
- To clean up our old caches so that they are not used going forward by the app, we should run the cleanup code only when we are certain that the old cache is not being used. The activate event listener is a good place to do this, since it will only run when the user opens a new page.
    - We can get an array of cache keys and use the Promise.all(...) method to ensure that all promises (keys turned into promises) finish up before moving on.
    - If the cache is not equal to the current cache name, then the cache would be deleted.

```
self.addEventListener('activate', function (event) {
  console.log('[Service Worker] Activating Service Worker ....', event);
  event.waitUntil(
    caches.keys()
      .then(function (keyList) {
        return Promise.all(keyList.map(function (key) {
          if (key !== CACHE_STATIC_NAME && key !== CACHE_DYNAMIC_NAME) {
            console.log('[Service Worker] Removing old cache.', key);
            return caches.delete(key);
          }
        }));
      })
  );
  return self.clients.claim();
});
```

- The dynamic and static caches should be assigned variable names at the top of the service worker file, so that new versions can be easily updated throughout the file.

```
var CACHE_STATIC_NAME = 'static-v4';
var CACHE_DYNAMIC_NAME = 'dynamic-v2';
```

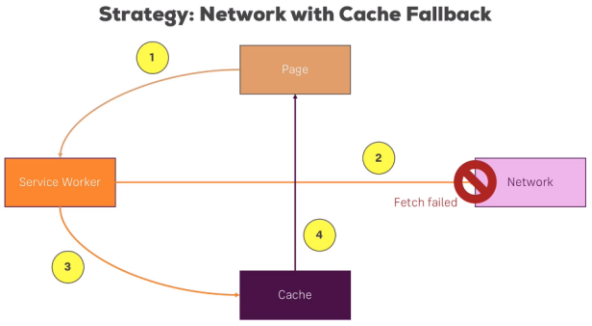
Section 6: Service Workers – Advanced Caching

General Information

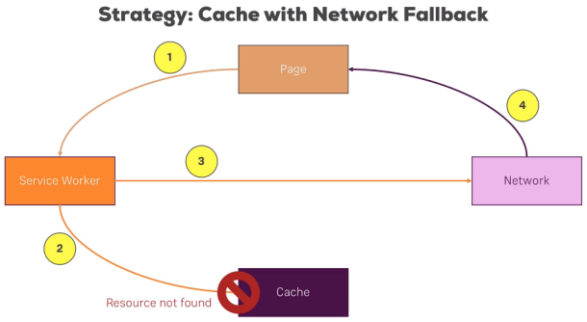
- We can access the cache storage (caches.open(...)) from the front-end as well as from the service worker.
    - Because of this, we can create a user-defined cache for items that the use decides to save into the cache from the front-end.
- It is ideal to create an offline fallback page that the user will be directed to if the user tries to load a page that is not cached.
    - Otherwise, the user would instead see the browser's page-not-found website, which would detract from the app-like feel of the PWA.

```
.catch(function (err) {
  return caches.open(CACHE_STATIC_NAME)
    .then(function (cache) {
      if (event.request.headers.get('accept').includes('text/html')) {
        return cache.match('/offline.html');
      }
    });
});
```

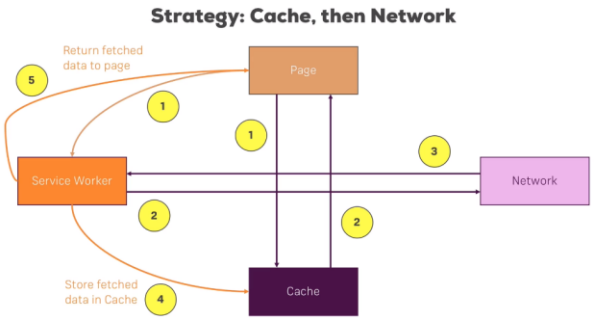
Caching Strategies



- A cache-only strategy would involve the app ignoring the network completely, even when online.
    - This is a strategy that is rarely used, but in certain situations, something like this could be helpful.
- Another strategy is the network-only strategy, where the application reaches out to the network through the service worker.
    - This strategy would completely ignore the cache, which is also not an ideal strategy for a PWA.
- The network with cache fallback strategy involves first checking with the network for a resource, and then reaching out to the cache only if the network is not available.
    - This seems great, but it can impact the responsiveness of the app since the network will cause a slowdown when loading pages, even if they have been previously cached.
    - A particularly bad case would be where the network times out after 30 seconds or something on a weak signal, before reaching out to the cache. It would be confusing and frustrating for the user experience.



- A strategy that does work well is the cache then network strategy.
    - This strategy first reaches out to the cache to load a page quickly and follows up with a network call to see if a more up-to-date page is available to override the cache version with.
    - We do have a variable that will store the network response and will use that in case the network response is quicker than loading from the cache.
    - While the page reaches to the cache, it will also reach out to the network via the service worker.
    - As shown in the diagram, the page (not the service worker) reaches out to the cache to load a page, as well as reaches out to the network via the service worker intercepting the fetch request.
    - When the network request returns, we then store it in the dynamic cache storage for next time and reload the page if it differs from the cache version.



- Ultimately, a hybrid approach is used where the main page that changes (e.g. the feed in Max's example) uses the cache then network approach, the other pages use the cache with network fallback approach, and the static pages that we define uses the cache only strategy.

Cleaning up the Cache

- The dynamic cache can fill up over time if cleaning logic is not put in place for the cache.

```
function trimCache(cacheName, maxItems) {
  caches.open(cacheName)
    .then(function (cache) {
      return cache.keys()
        .then(function (keys) {
          if (keys.length > maxItems) {
            cache.delete(keys[0])
              .then(trimCache(cacheName, maxItems));
          }
        });
    })
}
```

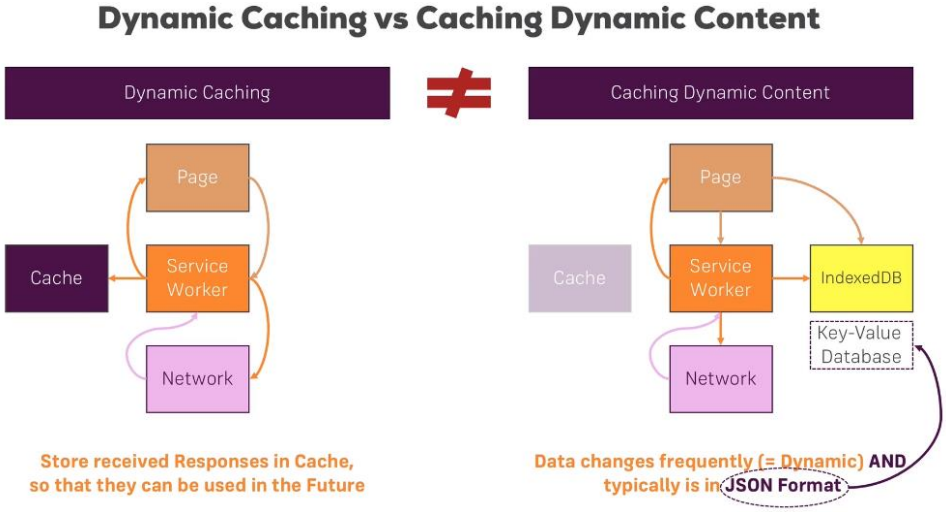
- We can put in place a recursive function for trimming the dynamic cache array, for example.
    - This would ensure that the oldest items in the dynamic cache are continuously deleted, keeping the dynamic cache size limited.
- Of course, a more elaborate function could be put in place to trim the cache based on what is considered the most important data to maintain.
    - This function can be called wherever we feel is most appropriate.
- Keep in mind that storage limits are in place for the cache, so we will have to have to be implemented.

```
function isInArray(string, array) {
  var cachePath;
  if (string.indexOf(self.origin) === 0) {
// request targets domain where we serve the page from (i.e. NOT a CDN)
    console.log('matched ', string);
    cachePath = string.substring(self.origin.length);
// take the part of the URL AFTER the domain (e.g. after localhost:8080)
  } else {
    cachePath = string; // store the full request (for CDNs)
  }
  return array.indexOf(cachePath) > -1;
}

self.addEventListener('fetch', function (event) {
  var url = 'https://httpbin.org/get';
  if (event.request.url.indexOf(url) > -1) {
    event.respondWith(
      caches.open(CACHE_DYNAMIC_NAME)
        .then(function (cache) {
          return fetch(event.request)
            .then(function (res) {
              // trimCache(CACHE_DYNAMIC_NAME, 3);
              cache.put(event.request, res.clone());
              return res;
            });
        })
    );
  } else if (isInArray(event.request.url, STATIC_FILES)) {
    event.respondWith(
      caches.match(event.request)
    );
  } else {
    event.respondWith(
      caches.match(event.request)
        .then(function (response) {
          if (response) {
            return response;
          } else {
            return fetch(event.request)
              .then(function (res) {
                return caches.open(CACHE_DYNAMIC_NAME)
                  .then(function (cache) {
                    // trimCache(CACHE_DYNAMIC_NAME, 3);
                    cache.put(event.request.url, res.clone());
                    return res;
                  })
              })
              .catch(function (err) {
                return caches.open(CACHE_STATIC_NAME)
                  .then(function (cache) {
                    if (event.request.headers.get('accept').includes('text/html')) {
                      return cache.match('/offline.html');
                    }
                  });
              });
          }
        })
    );
  }
});
```

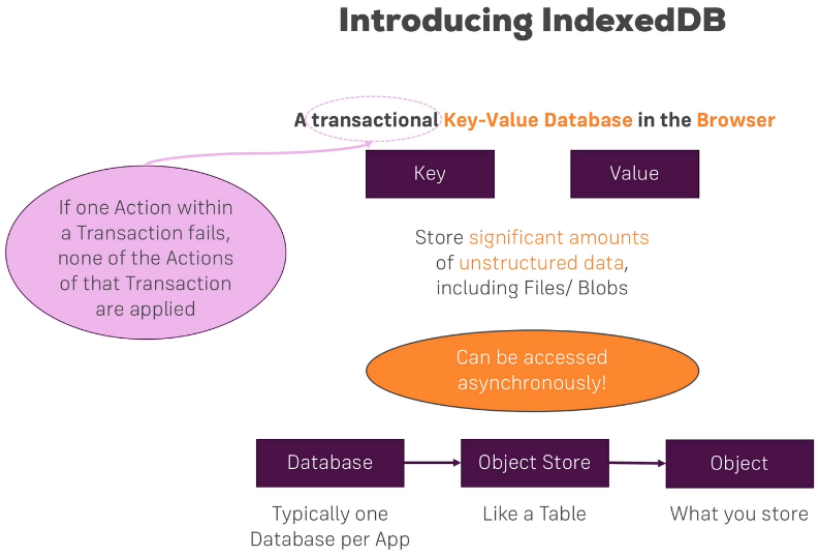
Section 7: IndexedDB and Dynamic Data

General Information

- IndexDB should be used for caching dynamic data from the server.
    - Firebase realtime databases can be used.



- You don't store CSS and files in IndexedDB (browser), but rather key values pair of information.

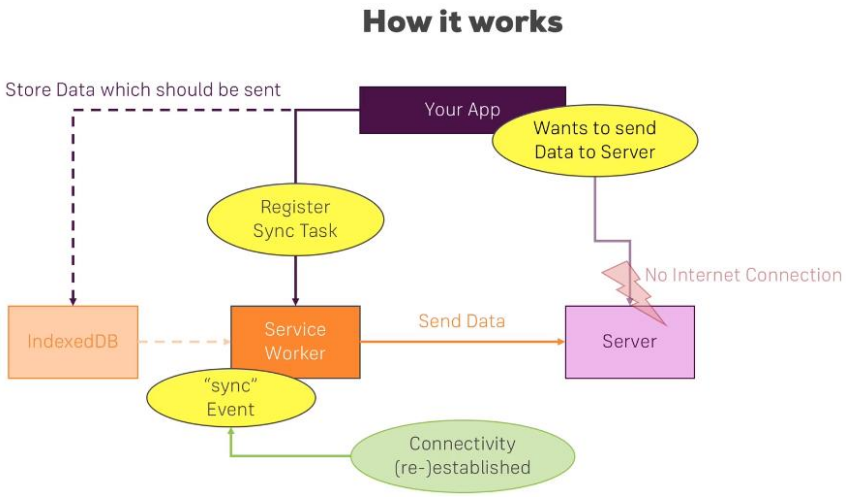


- IDB is a helpful package that we can use to make it easier to work with IndexedDB.
- We can view the IndexedDB for an application by going to Chrome Developer Tools ➔ Application ➔ IndexedDB.
- We need to handle the case where dynamic data was removed and update the IndexedDB accordingly.
    - One way to handle that is to clear and repopulate the IndexedDB each time that a network connection is made.
    - We could also delete single items instead.

Section 9: Background Sync

General Information

- Allows for manipulating and sending data while offline.
- This will even work if the browser is not open when connection is re-established.
- SyncManager is what is used perform this sync.

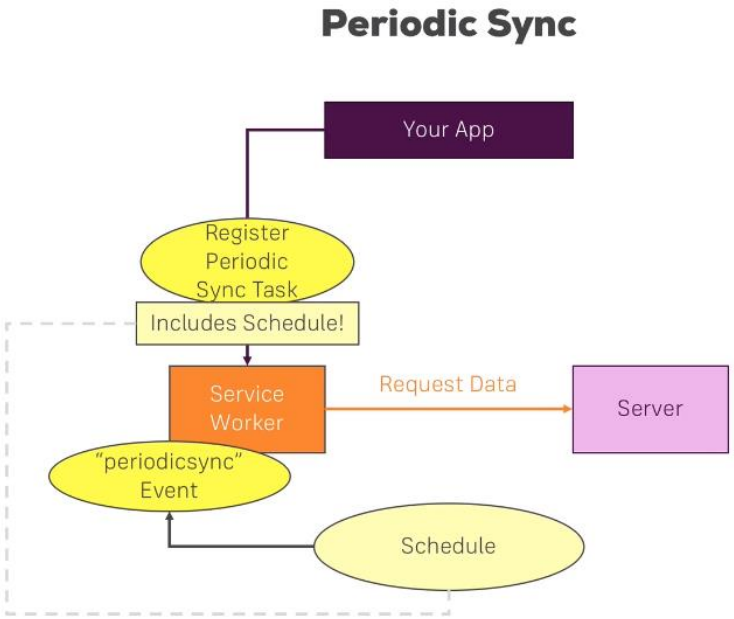


- Sw.sync.register(...) is what is needed to register the service worker with sync.
- We use the fetch(...) method to send data to the server if the browser doesn't support service workers or the sync manager.
    - This is a fall-back, which we should always consider.
- A self.addEventListener('sync', function() {...} event should run if a browser does support service workers and sync, and has confirmed that network connectivity has been re-established.
    - Once the data has been successfully sent to the server after connection is established, the data can be deleted from the indexedDB.

```
self.addEventListener('sync', function(event) {
  console.log('[Service Worker] Background syncing', event);
  if (event.tag === 'sync-new-posts') {
    console.log('[Service Worker] Syncing new Posts');
    event.waitUntil(
      readAllData('sync-posts')
        .then(function(data) {
          for (var dt of data) {
            fetch('https://pwagram-99adf.firebaseio.com/posts.json', {
              method: 'POST',
              headers: {
                'Content-Type': 'application/json',
                'Accept': 'application/json'
              },
              body: JSON.stringify({
                id: dt.id,
                title: dt.title,
                location: dt.location,
                image: 'https://firebasestorage.googleapis.com/v0/b/pwagram-99adf.appspot.com/o/sf-boat
.jpg?alt=media&token=19f4770c-fc8c-4882-92f1-62000ff06f16'
              })
            })
              .then(function(res) {
                console.log('Sent data', res);
                if (res.ok) {
                  deleteItemFromData('sync-posts', dt.id);
                }
              })
              .catch(function(err) {
                console.log('Error while sending data', err);
              });
          }
```

- You must turn off the Wi-Fi on your machine to test this.

- Periodic sync will allow data to be periodically fetched from the server in the background.

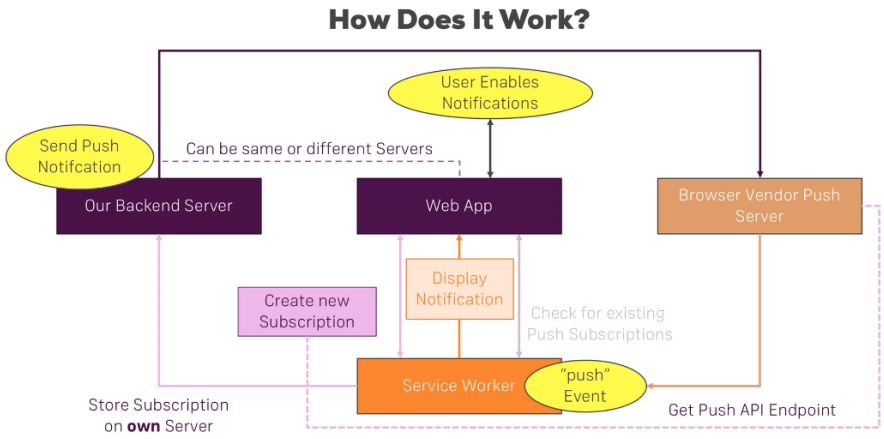


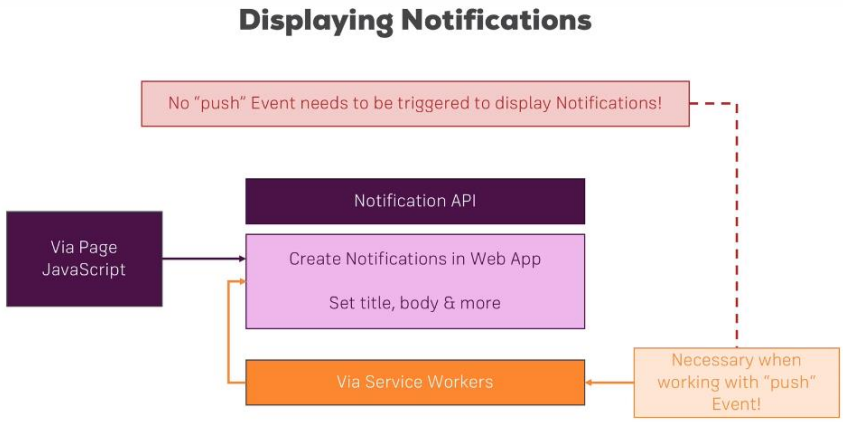
- Google Firebase allows you to make your own APIs.

Section 10: Notifications

General Information

- Notifications allow for another native feature on the device.





- We need to first request permissions from the user to display notifications.

Section 11: Native Device Features

General Information

- Location is another feature that can be used by PWAs.

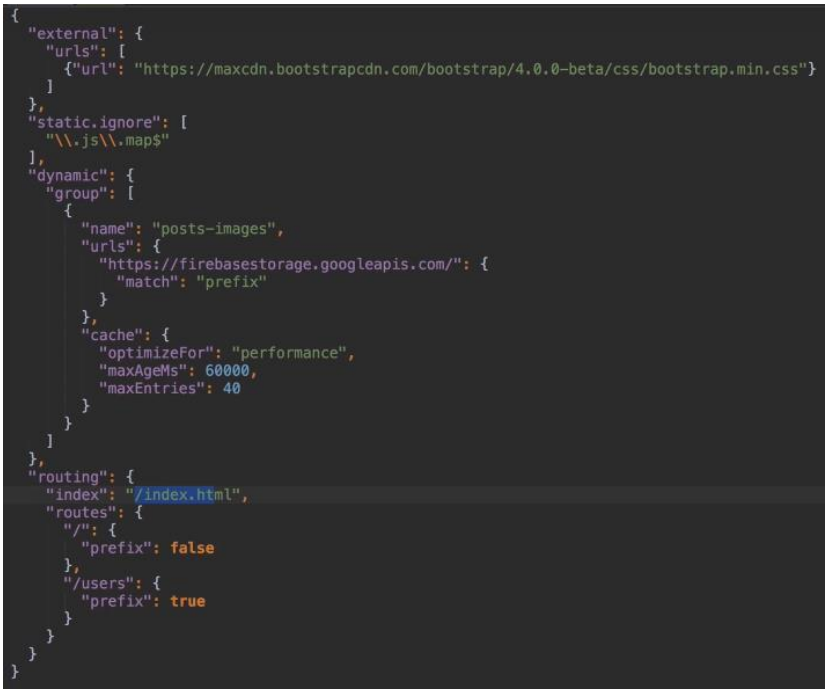
```
locationBtn.addEventListener('click', function (event) {
  if (!('geolocation' in navigator)) {
    return;
  }
  var sawAlert = false;

  locationBtn.style.display = 'none';
  locationLoader.style.display = 'block';

  navigator.geolocation.getCurrentPosition(function (position) {
    locationBtn.style.display = 'inline';
    locationLoader.style.display = 'none';
    fetchedLocation = {lat: position.coords.latitude, lng: 0};
    locationInput.value = 'In Munich';
    document.querySelector('#manual-location').classList.add('is-focused');
  }, function (err) {
    console.log(err);
    locationBtn.style.display = 'inline';
    locationLoader.style.display = 'none';
    if (!sawAlert) {
      alert('Couldn\'t fetch location, please enter manually!');
      sawAlert = true;
    }
    fetchedLocation = {lat: 0, lng: 0};
  }, {timeout: 7000});
});
```

Section 13: SPAs and PWAs

General Information

- React and Angular can be used for a PWA.
- We need to set the "serviceWorker": true tag in angular.json.

    - Service workers are only setup in the production build, so we must use ng build --prod in the command line.
- Cached data, including pages, are stored in the dist/ folder.
- We need to also add an Angular service worker manifest file as ngsw-manifest.json.
    - This would specify dependencies like external URLs that we should pre-cache.
    - It would also specify how to handle dynamic content, along with routes.



```
{
  "external": {
    "urls": [
      {"url": "https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css"}
    ]
  },
  "static.ignore": [
    "\\.js\\.map$"
  ],
  "dynamic": {
    "group": [
      {
        "name": "posts-images",
        "urls": {
          "https://firebasestorage.googleapis.com/": {
            "match": "prefix"
          }
        },
        "cache": {
          "optimizeFor": "performance",
          "maxAgeMs": 60000,
          "maxEntries": 40
        }
      }
    ]
  },
  "routing": {
    "index": "/index.html",
    "routes": {
      "/": {
        "prefix": false
      },
      "/users": {
        "prefix": true
      }
    }
  }
}
```

- We also need a manifest.json file so that we can get the app install banner.
    - A link to the manifest has to be added to manifest.json in the index.html.
- Max serves the PWA with ng serve --prod.