## Introduction to Progressive Web Applications 19th Day

Progressive Web Applications are experiences that combine the best of web applications and the best of mobile applications. They use service workers, HTTPS, a manifest file and an app shell architecture to deliver native app experiences to web applications.

A service worker is a script that your browser runs in the background, separate from the web page, opening the door to features that don't need a web page or user interaction. They include features such as push notifications and background sync.

Progressive Web Apps are also described as user experiences that have the reach of the web, and are largely characterized by the following:

- **Reliable** – Load instantly and never show the "No Internet Connection" page, even in uncertain network conditions.
- **Fast** – Respond quickly to user interactions with silky smooth animations and no janky scrolling.
- **Engaging** – Feel like a natural app on the device, with an immersive user experience.

### Advantages of Progressive Web Apps.

The advantages of Progressive Web Apps (PWA) are massive and I'll highlight them below:

- **Cost** – The cost of building a PWA is less than that of a mobile application.
- **Progressive** – Works for every user, regardless of browser choice because they're built with progressive enhancement as a core tenet.
- **Responsive** – Fit any form factor: desktop, mobile, tablet, or forms yet to emerge.
- **Connectivity independent** – Service workers allow apps to work offline or on low-quality networks.

- **App-like** – Feel like a native app to the user with app-style interactions and navigation.
- **Fresh** – Always up-to-date thanks to the service worker update process.
- **Safe** – Served via HTTPS to prevent snooping and ensure content hasn't been tampered with.
- **Easy Discovery** – Are identifiable as "applications" thanks to W3C manifests and service worker registration scope allowing search engines to find them.
- **Re-engageable** – Make re-engagement easy through features such as push notifications.
- **Installable** – Allow users to "keep" apps they find most useful on their home screen without the hassle of an app store.

**Building blocks of a progressive web application**

Progressive Web Apps are characterized by a Service Worker file, a Web App Manifest, HTTPS, Push Notifications and Background Sync.

**Service workers**.

A service worker is a script that your browser runs in the background, separate from the web page, opening the door to features that don't need a web page or user interaction. Service workers are installed on the device from which the website is accessed from and they allow you to control how network requests from your page are handled.

A typical Service Worker process goes like this Register → Install → Fetch → Activation

1)Register.

To install a service worker for your site you need to register it, which you do in your page's JavaScript. Registering a service worker will cause the browser to start the service worker install step in the background.

Example:

```
// Checks if the browser supports Service workers
    if ('serviceWorker' in navigator) {

/* registerServiceWorker.js is the file that contains the install, fetch and
activativation code. */

      navigator.serviceWorker.register('registerServiceWorker.js')
      .then(registration => {
 // Successful registration
        console.log('Hooray. Registration successful, scope is:',
registration.scope);
      }).catch(function(err) {
// Failed registration, service worker won't be installed
        console.log('Oops. Service worker could not be installed, error:',
error);
      });
    }
```

The registerServiceWorker.js file referenced in the code block above usually contains the install, fetch and activation code.

Typically, during the install step, you'll want to cache some static assets. If all the files are cached successfully, then the service worker becomes installed. If any of the files fail to download and cache, then the install step will fail and the service worker won't activate (i.e. won't be installed).

```
var CACHE_NAME = 'pwa-cache-v1';
    var urlsToCache = [
      '/',
      '/styles/styles.css',
      '/script/webpack-bundle.js'
    ];
    self.addEventListener('install', event => {
      event.waitUntil(
        caches.open(CACHE_NAME)
          .then(cache => {
            // Open a cache and cache our files
            return cache.addAll(urlsToCache);
          })
      );
    });
```

The urlsToCache is an array of the files to be cached. The cache.addAll() function requests, fetches them and adds them to the cache.

Now that the service worker has been installed and resources are being cached, it's expected that the next thing to be done is getting all the cached resources.

Whenever a user makes a request (navigates to a page or refreshes a page), the service worker fetch event is called and this method checks the request and finds any cached results from any of the caches your service worker created.

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

In the code block above, the fetch event is defined, and in the event.respondWith() function we pass in a promise from caches.match(). The caches.match() method looks at the request and finds any cached results from any of the caches your service worker created.

If there is a matching response, the cached value is returned. Otherwise, we return the result of a call to fetch, which will make a network request and return the data if anything can be retrieved from the network.

The activate event is usually used for cache management. It is triggered after registering, and it is also used to clean up old caches for newer ones.

```
self.addEventListener('activate', function(event) {
  var cacheWhitelist = ['pages-cache-v1', 'blog-posts-cache-v1'];
  event.waitUntil(
    // Get all the cache keys (cacheName)
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        // Check all caches and delete old caches that are not whitelisted
        cacheNames.map(function(cacheName) {
          if (cacheWhitelist.indexOf(cacheName) === -1) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

The code block above is an example of an activate event. It loops through all of the caches in the service worker and deletes any caches that aren't defined in the cache whitelist.

After the activation step, the service worker will control all pages that fall under its scope, though the page that registered the service worker for the first time won't be controlled until it's loaded again.

### Web app manifest

A web app manifest controls what the user sees when launching from the home screen. This includes things like a splash screen, theme colors, and even the URL that has opened.  It is a json file.

### manifest.json

```
{
```

```
  "short_name": "Lux App",
  "name": "Lux",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "android-chrome-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ],
  "start_url": "./index.html",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

- A short_name is required for the text on the user's home screen.
- A name is required for use in the Web App Install banner.
- The icons are used when a user adds your site to their home screen. You can also define a set of icons for the browser to use.
- The start_url specifies the URL that loads when a user launches the application from a device. If given as a relative URL, the base URL will be the URL of the manifest.
- The display defines the developer's preferred display mode for the web application. It could be standalone, fullscreen, browser, or minimal-ui.
- The theme_color defines the default theme color for an application. This sometimes affects how the application is displayed by the OS (e.g., on Android's task switcher, the theme color surrounds the application).
- The background_color  defines the expected background color for the web application.

## HTTPS

It's very important that Progressive Web Apps be served from a secure origin. That's why it's a requirement for an application to be served with HTTPS so it can be considered a Progressive Web App. Using HTTPS also ensures that intruders can't tamper with the communications between your websites and your users' browsers.

## Background sync

Background sync is a web API that lets you defer actions until the user has stable connectivity. This is useful for ensuring that whatever the user wants to send is actually sent, even after the loss of internet connectivity.

As an example, background sync can be very useful in a chat application. I'm sure you've witnessed situations where after a message was sent with a bad connection or no connection at all, it still goes on to deliver the message after a good connection has been established. This is background sync at work.

Implementing background sync in a PWA is very straightforward. Take the code below as an example:

```
// Register your service worker (service worker with the name sw.js) :
  navigator.serviceWorker.register('/sw.js');

// Then later, request a one-off sync:
  navigator.serviceWorker.ready.then(function(swRegistration) {
    return swRegistration.sync.register('myFirstSync');
  });
```

In the code above, the service worker is registered and then we request a one-off sync with the name of myFirstSync. Then we listen for the event in /sw.js:

```
self.addEventListener('sync', function(event) {
        if (event.tag == 'myFirstSync') {
          event.waitUntil(doSomeStuff());
        }
      });
```

In the code block above, doSomeStuff() should return a promise indicating the success/failure of whatever it's trying to do. If it fulfills, the sync is complete. If it fails, another sync will be scheduled to retry. Retry syncs also wait for connectivity and employ an exponential back–off.

The name of the sync should be unique for a given sync. If you register for a sync using the same tag as a pending sync, it merges with the existing one.

**Push notifications**.

Web Push Notifications are simply a way of allowing users to opt in to timely updates from the sites they love and also allow you to effectively re–engage them with customized, relevant content. They work with service workers because of the background usage of service workers.

It's important to note that Push and Notification use different, but complementary, APIs. **Push** is invoked when a server supplies information to a service worker; a **notification** is the action of a service worker or web page script showing information to a user. Therefore, in order for a push notification to work, both a server and a client are needed.

Progressive Web Apps also rely on some patterns and technologies that help deliver a meaningful user experience regardless of the quality of internet connectivity. I'll highlight some below.

**PRPL**.

The **P**ush **R**ender **P**re–cache **L**oad pattern is a relatively new and experimental pattern that takes advantage of modern web platform features

(service workers) to granularly deliver mobile web experiences more quickly.

It is simply a pattern for structuring and serving Progressive Web Apps, with an emphasis on the performance of app delivery and launch. It stands for:

- **Push** critical resources for the initial URL route.
- **Render** initial route.
- **Pre-cache** remaining routes.
- **Lazy-load** and create remaining routes on demand.

The PRPL pattern is about making sure an application is built in a way that the user gets the best mobile experience by having the lowest possible minimum time-to-interactive (especially on first use) and the best maximum caching efficiency (caching routes and resources).

**Web storage**

Offline support is a key feature of a Progressive Web app, and for offline support to work, some sort of data persistence is needed. This is where Web Storage comes in.

Web Storage allows developers to cater for instances whereby a user temporarily loses internet connectivity. In order to make the Progressive Web App still usable in an instance like that, web storage can be used to display already saved data when needed.

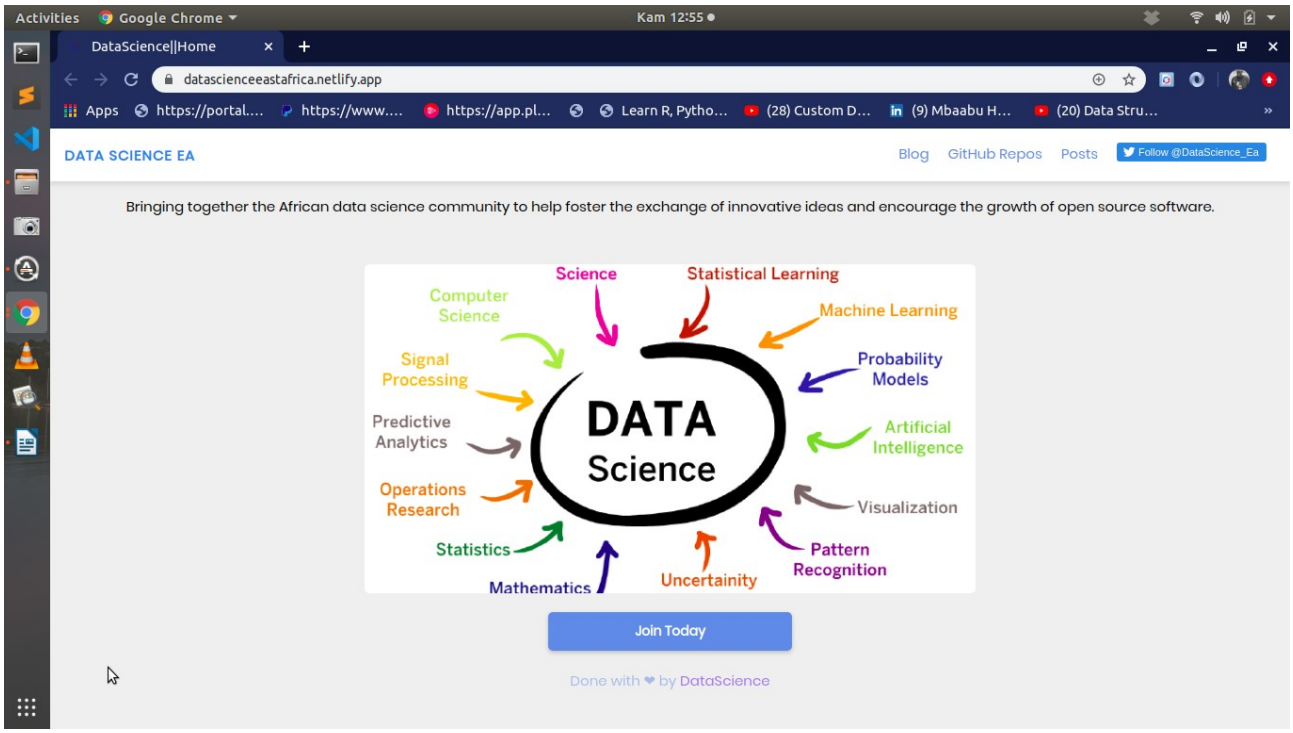
There are various Web Storage APIs available such as Local Storage, Session Storage, and IndexedDB and each comes with its own pros and cons.

Example:

Data Science East Africa app:

Link: https://datascienceeastafrica.netlify.app/

Code: https://github.com/HarunHM/DataScience-East-Africa-UI




More material:

https://developers.google.com/web/ilt/pwa

https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction

Best Wishes

Lux Tech Academy