PWA workshop

Part 3

Keeping your data synchronized with BackgroundSync API

Using Keeping your data synchronized

1. Introduction

When I'm trying to access information on the go on my mobile phone, nothing is more frustrating than not being able to get to that information I need so desperately, especially when I know that I've viewed a certain web page before. Fortunately, you can build Progressive Web Apps (PWAs) to deal with such situations. In this code lab, you'll learn how to build web apps that can work offline and deal with situations where you may be in an area with poor or no network coverage at all.

What You'll Learn

- ✓ How to use the power of Service Workers to build resilient web apps that work with no connection
- ✓ How to build PWAs that cater to situations where the user has a network connection, but it's slow, flaky, or prone to drop occasionally
- ✓ Take the offline web a step further and look at a new web API called BackgroundSync
- ✓ Dive into a soon-to-be-released feature called `PeriodicSync`

2. Getting set up

Project Set Up

In this code lab, we are building on top of the project started in the `Service Worker Management with WorkBox` code lab.

If you didn't do it already: **Fork** and then **Clone** the following repository: `https://github.com/The-Guide/fe-guild-2019-pwa.git`

```
$ git clone https://github.com/[YOUR GITHUB PROFILE]/fe-guild-2019-pwa.git
$ cd fe-guild-2019-pwa
```

If you want to start directly with `Service Workers - Keeping your data synchronized` checkout the following branch:

```
$ git checkout pwa-sw-advanced-init
```

First install the dependencies

```
$ npm install
```

Then type in the terminal

```
$ npm start
```

and open Chrome at `localhost:8080fe-guild-2019-pwa/`

In this code lab, we are also going to connect the front end with the backend (**FINALLY!!!**).

Fork and then **Clone** the following repository: `https://github.com/The-Guide/fe-guild-2019-pwa-server.git`

```
$ git clone https://github.com/[YOUR GITHUB PROFILE]/fe-guild-2019-pwa-server.git
$ cd fe-guild-2019-pwa-server
```

Install dependencies

```
$ npm install
```

To start the project type in the terminal:

```
$ npm start
```

the server will be hosted at `localhost:3000`

3. Taking and displaying selfies

In **index.html** inside the **div** with **id *create-post***, there is a form tag. We are going to use it to submit the data. Don't worry about the picture for the moment; the server will give us a dummy one back. In **feed.js**:

```
const form = document.querySelector('form');
const titleInput = document.querySelector('#title');
const locationInput = document.querySelector('#location');

const API_URL = 'http://localhost:3000/selfies';

form.addEventListener('submit', event => {
   event.preventDefault();

   if (titleInput.value.trim() === '' || locationInput.value.trim() === '') {
       // Very professional validation
       alert('Please enter valid data!');
       return;
   }

   closeCreatePostModal();

   const id = new Date().getTime();
   const postData = new FormData();
   postData.append('id', id);
   postData.append('title', titleInput.value);
   postData.append('location', locationInput.value);

   fetch(API_URL, {method: 'POST', body: postData})
       .then(response => {
           console.log('Sent data', response);
       });
});
```

Explanation

1. First, we declare a reference to the tags of interest form, title, and location text boxes.
2. Then we declare the API_URL
3. Lastly, we attach an event listener for the submit event of the form. We do a simple fetch to POST the form data

Exercises

1. Update the feed.js with the code above
2. Don't forget to run npm run build to update the sw.js. Because we changed the feed.js file it's revision will also change inside sw.js
3. Make sure the new service worker is active:
    - Nuke version: Application -> Clear storage and then refresh
    - Best version: Refresh the page and skipInstall on the Service Worker
    - Ain't nobody got time for that version: Refresh the page, close the tab and open it back again
4. You can navigate to localhost:3000/selfies to admire yourself

Displaying selfies

So far we are sending data to the backend, but we don't display anything in the frontend. Let's change that

We will display the selfies as cards with picture, title, and location but first let read the selfies from the server.

In **feed.js**:

```
fetch(API_URL)
    .then(response=> response.json())
    .then(data => {
        console.log('From server', data);
    });
```

If you follow all the above procedures to refresh the Service Worker, then you should see in Console the data retrieved from the server. It is not an array but an object with the ids of each selfie as a key. We need to convert this to an array and update the UI

```
fetch(API_URL)
    .then(response=> response.json())
    .then(data => {
        console.log('From server', data);
        const selfies = [];
        for (const key in data) {
            selfies.push(data[key]);
        }
        updateUI(selfies);
    });
```

There is no updateUI function yet so let's write this next:

```
const updateUI = selfies => {
 clearCards();
 selfies.forEach(selfie => createCard(selfie));
};
```

Now there is an updateUI function, but we just introduced two additional ones. clearCards() will clear all the previously created cards for us

```
const sharedMomentsArea = document.querySelector('#shared-moments');

const clearCards = () => {
   while (sharedMomentsArea.hasChildNodes()) {
       sharedMomentsArea.removeChild(sharedMomentsArea.lastChild);
   }
};
```

The parent for the selfie cards will be a div with the id shared-moments, so we save a reference to it. Then inside the clearCards() function, we remove all the children form it.

createCard(selfie) will create the HTML content of the card and add it to the parent

```
const createCard = selfie => {
 const cardWrapper = document.createElement('div');
 cardWrapper.className = 'shared-moment-card mdl-card mdl-shadow--2dp';

 const cardTitle = document.createElement('div');
 cardTitle.className = 'mdl-card__title';
 cardTitle.style.backgroundImage = 'url(' + selfie.selfieUrl + ')';
 cardTitle.style.backgroundSize = 'cover';
 cardWrapper.appendChild(cardTitle);

 const cardTitleTextElement = document.createElement('h2');
 cardTitleTextElement.style.color = 'white';
 cardTitleTextElement.className = 'mdl-card__title-text';
 cardTitleTextElement.textContent = selfie.title;
 cardTitle.appendChild(cardTitleTextElement);

 const cardSupportingText = document.createElement('div');
 cardSupportingText.className = 'mdl-card__supporting-text';
 cardSupportingText.textContent = selfie.location;
 cardSupportingText.style.textAlign = 'center';
 cardWrapper.appendChild(cardSupportingText);

 // Material design lite stuff
 componentHandler.upgradeElement(cardWrapper);

 sharedMomentsArea.appendChild(cardWrapper);
};
```

Refresh (with all the ceremonies), and you should see something...clearly in need of some CSS.

In **feed.css**:

```
.shared-moment-card.mdl-card {
 margin: 10px auto;
 width: 80%;
}

@media (min-width: 600px) {
 .shared-moment-card.mdl-card {
     width: 60%;
 }
}

@media (min-width: 1000px) {
 .shared-moment-card.mdl-card {
     width: 45%;
 }
}

.shared-moment-card .mdl-card__title {
 height: 140px;
}

@media (min-height: 700px) {
 .shared-moment-card .mdl-card__title {
     height: 160px;
 }
}

@media (min-height: 1000px) {
 .shared-moment-card .mdl-card__title {
     height: 200px;
 }
}
```

Testing on mobile

Deploying the server or using a `serverless` solution is the subject for another code lab, so we have two options:

- **Hot spot**: This should work for everyone. Simply create a hot spot on your phone and connect your laptop's WiFi to it. If you restart, the application `http-server` will show an **Available on** followed by a list of URLs where the application can be accessed.

  Change the `API_URL` to that address, build and refresh on mobile. You should be able to post and see the pictures of the posts that you made on mobile. Any older content will be under `localhost:3000` and will not show any image.

- **Remote debugging via USB cable**: This will work everywhere for `Android` phone users (in combination with Chrome and only on Mac with Safari for `iPhone` users. While connected simply open the app on the phone.

Talking selfies

In double quotes (") because we are going to use a file input for the moment and not the camera (Beyond PWA code lab) but on mobile, you should get the option to take a photo and use that as a file.

In index.html just above the form tag:

```
<div id="pick-image">
   <h6>Pick an Image instead</h6>
   <input type="file" accept="image/*" id="image-picker">
</div>
```

The file input will be our fallback if we cannot use the camera, so we already prepare the terrain. Now back in **feed.js**:

```
const imagePicker = document.querySelector('#image-picker');
let picture;

imagePicker.addEventListener('change', event => picture = event.target.files[0]);
```

Explanation

1. Save a reference to the file input
2. We will save the image (from disk or camera) in the picture variable
3. Add an event handler for the change event of the file input and set the uploaded file to the picture variable.

Exercises

1. Update the application with the code above
2. Test on both desktop and mobile (if you are able to)
3. On mobile take a picture instead of uploading a file.

4. Browsing selfies offline

So far we can take selfies and see our weird duck faces, but we cannot admire ourselves offline. It is high time to fix this.

We are going to use IndexedDB to cache our selfies and to enable that we need to add some utilities.

1. Get **idb.js** via the command: **npm i --save idb** and copy the node_modules/idb/build/idb.js in: src/lib/idb.js
2. Create an empty utility.js file inside the src/js folder (*we will write code for it later*)
3. Make sure you reference the two files:
   In index.html

```
<script defer src="src/lib/material.min.js"></script>
<script src="src/lib/idb.js"></script>
<script src="src/js/utility.js"></script>
<script src="src/js/app.js"></script>
<script src="src/js/feed.js"></script>
```

In sw-template.js:

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/3.5.0/workbox-sw.js');
importScripts('src/lib/idb.js');
importScripts('src/js/utility.js');
```

utility.js

This file will have the code needed by both the Service Worker and the app itself

Start by moving the API_URL and add a little twist

```
// TODO: Change this with your own local IP (either localhost/127.0.0.1)
// or the IP assigned by the phone hot spot
const SERVER_URL = 'http://192.168.1.162:3000';
const API_URL = `${SERVER_URL}/selfies`;
```

Next, we need code to manage an IndexedDB database and stores.
We want to open the database, write data to a store, read all data, delete all data or delete just one item.

```
const dbPromise = idb.openDb('selfies-store', 1, upgradeDB => {
    if (!upgradeDB.objectStoreNames.contains('selfies')) {
        upgradeDB.createObjectStore('selfies', {keyPath: 'id'});
    }
});

const writeData = (storeName, data) => {
    return dbPromise
        .then(db => {
            const tx = db.transaction(storeName, 'readwrite');
            const store = tx.objectStore(storeName);
            store.put(data);
            return tx.complete;
        });
};

const readAllData = storeName => {
    return dbPromise
        .then(db => {
            const tx = db.transaction(storeName, 'readonly');
            const store = tx.objectStore(storeName);
            return store.getAll();
        });
};

const clearAllData = storeName => {
    return dbPromise
        .then(db => {
            const tx = db.transaction(storeName, 'readwrite');
            const store = tx.objectStore(storeName);
            store.clear();
            return tx.complete;
        });
};

const deleteItemFromData = (storeName, id) => {
    dbPromise
        .then(db => {
            const tx = db.transaction(storeName, 'readwrite');
            const store = tx.objectStore(storeName);
            store.delete(id);
            return tx.complete;
        })
```

sw-template.js

Let's first cache the selfies as they arrive from the server. For that, we need to register a route in sw-template.js

```
workbox.routing.registerRoute(API_URL, args => {
 return fetch(args.event.request)
   .then(response => {
     const clonedResponse = response.clone();
     clearAllData('selfies')
       .then(() => clonedResponse.json())
       .then(selfies => {
         for (const selfie in selfies) {
           writeData('selfies', selfies[selfie])
         }
       });
     return response;
   });
});
```

We register a route for the API_URL, and if the fetch is successful, we call the clearAllData function to delete all entries from the selfies store. After everything is deleted, we call writeData to save the fresh data from the server and return the response back to the UI.

We now need to get the selfies from the IndexedDB store in case we don't receive anything from the network.

In feed.js adjust the fetch for selfies like this:

```
let networkDataReceived = false;
fetch(API_URL)
 .then(response => response.json())
 .then(data => {
   console.log('From server', data);
   networkDataReceived = true;
   const selfies = [];
   for (const key in data) {
     selfies.push(data[key]);
   }
   updateUI(selfies);
 });
```

Notice the presence of the networkDataReceived variable. By default, it is set to false, but when we get data, we set it to true.

Next, add code to read the selfies from IndexedDB if networkDataReceived is false

```
if ('indexedDB' in window) {
 readAllData('selfies')
   .then(selfies => {
     if (!networkDataReceived) {
       console.log('From cache', selfies);
       updateUI(selfies);
     }
   });
}
```

Explanation

We first check to see if IndexedDB is supported. Notice that we didn't check for it in the Service Worker. If the broswer supports service workers will also support IndexedDB. Then we read the data from the selfies store by calling the readAllData function, and if networkDataReceived is false, we update the UI.

Exercises

1. Implement and experiment with the code above
2. Try to take your app offline and manually add selfies. Either directly in the data/selfies.json on the server application or by using POSTMAN
3. Try to see what happens if you have a slow network (adjust it from the Network tab in Chrome Developer Tools
4. While offline you can only see the title and location of the selfies, but no image. See if you can add a route in the service worker for it.

5. Taking selfies offline

The BackgroundSync API allows users to queue data that needs to be sent to the server while a user is working offline, and then as soon as they're online again, it sends the queued data to the server. This is useful for when you want to ensure that what your user submits to the server truly gets sent. To give you a quick example of this on a practical level, say a user needs to be able to edit the details of a blog post using a Content Management System (CMS). If the CMS uses Service Workers and BackgroundSync, the user can edit the contents of the blog post offline, and then the CMS will sync the results when the user is online again. This functionality allows users to work on the go, regardless of whether they're connected to the internet.

So far, we've been focusing on building websites that can function when the user is offline and dealing with situations where unreliable networks can cause failures. This functionality is great, but until now most of these pages have been read-only—you're only loading web pages and displaying information. What if you wanted the user to send something to the server while the user is offline? For example, they may want to save something important using their web app, safe in the knowledge that when they re-establish a network connection, their important information will be sent through to the server. BackgroundSync was built to handle that scenario.

BackgroundSync is a new web API that lets you defer actions until the user has stable connectivity, which makes it great for ensuring that whatever the user wants to send is sent when they regain connectivity. For example, let's say someone using the Progressive Selfies web app wants to take a selfie in the perfect spot but there is no signal, and the app is offline. With BackgroundSync, they can "send" can do that while offline, and once they regain connectivity, the Service Worker will send the data in the background.

Sync selfies

To apply BackgroundSync to our app, we need to create a store in our IndexedDB database to hold our *"to be synced selfies"*.

We do that in utility.js

```
const dbPromise = idb.openDb('selfies-store', 1, upgradeDB => {
 if (!upgradeDB.objectStoreNames.contains('selfies')) {
   upgradeDB.createObjectStore('selfies', {keyPath: 'id'});
 }

 if (!upgradeDB.objectStoreNames.contains('sync-selfies')) {
   upgradeDB.createObjectStore('sync-selfies', {keyPath: 'id'});
 }
});
```

Next, we need to change the way we send data to the server. We will always use BackgroundSync unless the browser doesn't support it. **In feed.js**

```
form.addEventListener('submit', event => {
 event.preventDefault();
 if (titleInput.value.trim() === '' || locationInput.value.trim() === '') {
  // Very professional validation
  alert('Please enter valid data!');
  return;
 }
 closeCreatePostModal();
const id = new Date().getTime();
```

```
if ('serviceWorker' in navigator && 'SyncManager' in window) {
  navigator.serviceWorker.ready
    .then(sw => {
      const selfie = {
        id: id,
        title: titleInput.value,
        location: locationInput.value,
        selfie: picture,
      };
      writeData('sync-selfies', selfie)
        .then(() => sw.sync.register('sync-new-selfies'))
        .then(() => {
          const snackbarContainer = document.querySelector('#confirmation-toast');
          const data = {message: 'Your Selfie was saved for syncing!'};
          snackbarContainer.MaterialSnackbar.showSnackbar(data);
        })
        .catch(function (err) {
          console.log(err);
        });
    });
```

```
} else {
  const postData = new FormData();
  postData.append('id', id);
  postData.append('title', titleInput.value);
  postData.append('location', locationInput.value);
  postData.append('selfie', picture);

  fetch(API_URL, {method: 'POST', body: postData})
    .then(response => console.log('Sent data', response));
 }
```

Explanation

Let's start with the obvious: the first part of the code including the generation of the id is the same.

First, we do a simple check to see if the browser supports Service Workers. If it does, and if the Service Worker is ready, register a sync with the tag sync-new-posts. This is a simple string that is used to recognize this event. You can think of these sync tags as simple labels for different actions. You can have as many as you want.

We're registering a sync using the registration object and providing it with a tag to identify it. Each sync must have a unique tag name because if we register a sync using the same tag as a pending sync, they will combine together. If the user tries to send seven messages while offline, they'll only get one sync when they regain connectivity. If you did want this to happen seven times, you need to use seven unique tag names.

Finally, we retrieve the values from the input fields and save them into the IndexedDB. With these values stored safely in the IndexedDB, you can retrieve them when the sync event takes place in the Service Worker.

If everything was successful, we show a toast message to the user.

The Service Worker

Before BackgroundSync functions correctly, you need to update the Service Worker code. The following listing contains the code that will respond to our newly created sync event. In **sw-template.js**

```
self.addEventListener('sync', event => {
 console.log('[Service Worker] Background syncing', event);
 if (event.tag === 'sync-new-selfies') {
  console.log('[Service Worker] Syncing new Posts');
  event.waitUntil(
   readAllData('sync-selfies')
    .then(syncSelfies => {
     for (const syncSelfie of syncSelfies) {
      const postData = new FormData();
      postData.append('id', syncSelfie.id);
      postData.append('title', syncSelfie.title);
      postData.append('location', syncSelfie.location);
      postData.append('selfie', syncSelfie.selfie);
…..
```

```
        fetch(API_URL, {method: 'POST', body: postData})
          .then(response => {
            console.log('Sent data', response);
            if (response.ok) {
              response.json()
                .then(resData => {
                  deleteItemFromData('sync-selfies', parseInt(resData.id));
                });
            }
          })
          .catch(error => console.log('Error while sending data', error));
      }
    })
  );
 }
});
```

Explanation

The listing above adds an event listener for the sync event. This event will only fire when the browser believes that the user has connectivity. You may also notice the check to confirm that the current event has a tag that matches the string 'sync-new-selfies'. This tag was added to the submit listener. If we didn't have this tag, the sync event would fire every time the user had connectivity and process your logic repeatedly.

Next, we retrieve the payload values that were stored in the IndexedDB when the user clicked the submit button. With these values, we then use the fetch API to POST the values to the server. The last step in the logic is to clean up afterwards and remove the values that are stored in the IndexedDB to ensure that you don’t have any old data lying around.

If all these steps were successful, the fetch request would return a successful result. If for any reason the fetch request wasn’t successful, the BackgroundSync API will try again. BackgroundSync has some clever retry functionality built into it to deal with a situation where the promise might fail.

Like most Service Worker–based code, BackgroundSync expects a promise because it needs to signal to the browser that the sync event is ongoing, and it needs to keep the Service Worker active if possible. If for any reason the fetch request failed and it received a promise that rejected, it will signal the browser that the sync failed, and this will cause the browser to reschedule the event. This functionality is handy when you want to ensure that what your user submits gets sent.

Under the hood, the browser might combine syncs together to reduce the number of times that the current device, network connection (radio), and browser need to wake up. Although these event timings may be combined, you still get a new event per pending sync.

Testing

Believe it or not, testing all this is easier than you think: once you've visited the page and your Service Worker is active, all you need to do is disconnect from the network by unplugging the network cable, disabling your Wi-Fi, or changing your network connection using the Developer Tools.

6. Periodic synchronization

Imagine the following scenario: a user opens up their phone to see that they already have the latest selfies for the Progressive Selfies App—which is strange because they're currently offline and haven't visited the web app today. Instead, a sync happened in the background while they were sleeping. New data was synced to their phone before they even woke up and was available for them in an instant. Very impressive!

This feature, known as PeriodicSync, allows you to schedule a sync for a predetermined time. It's simple to set up, doesn't require any server configuration, and allows the browser to optimize when it fires in order to be helpful and less disruptive to the user.

At the time of writing this code lab, PeriodicSync is still being developed (and is therefore subject to change), but it will be available in browsers shortly. It is powerful functionality that's worth sharing, which is why I wanted to include it here at this early stage. The following listing gives you an idea of what this code might look like when it is released.

```
if ('serviceWorker' in navigator && 'SyncManager' in window) {
 navigator.serviceWorker.ready.then(sw => {
   sw.periodicSync.register({
     tag: 'get-latest-selfies',
     minPeriod: 12 * 60 * 60 * 1000,
     powerState: 'avoid-draining',
     networkState: 'avoid-cellular'
   })
     .then(periodicSyncReg => console.log('Success!'))
     .catch(error => console.log('Error', error))
 });
}
```

This code is similar to the code for registering BackgroundSync, except you're registering a PeriodicSync.Similar to BackgroundSync, you need to register the sync with a tag name in order to identify how to respond accordingly, and much like BackgroundSync, each tag name needs to be unique to ensure that a different action takes place.

You'll notice that the PeriodicSync API also accepts a value called minPeriod. This value is used to determine the minimum time between sync events and is set in milliseconds.If you set the value to 0, it will allow the browser to fire the event as frequently as it wants.

Because syncs will run repeatedly, it's important that the PeriodicSync API take into account the battery and network state of the device it's running on. As developers, we need to be responsible to our users and not drain their battery or generate hefty mobile bills. Configuring properties such as powerState can avoid such events because they can either be set to 'auto' or 'avoid-draining'. 'auto' allows syncs to occur during battery drain, but it may be restricted if the device has battery-saving mode enabled. 'avoid-draining' will delay syncs on battery-powered devices while the battery isn't charging.

You can also determine the network usage of a device by configuring the networkState property. By setting the value to 'avoid-cellular', the browser will delay syncs while the device is connected to a cellular network. 'online' will delay syncs if the device is online, and 'any' is similar to 'online', except syncs may happen while the device is offline.

It’s worth noting that PeriodicSync isn't meant to be an exact timer. Although the API accepts a minPeriod in milliseconds, it could mean that the sync might not fire exactly on time. All this could be due to the network connection, battery state, or the settings of the current device. Due to the nature of PeriodicSync requiring device resources, it’s highly likely that it will require opt-in permission from the user.

Solution

Source code

```
$ git checkout pwa-sw-advanced-final
```