Table of Contents

Progressive Web Apps ILT - Codelabs

Curriculum for instructor-led training course for teaching Progressive Web Apps concepts. Developed by Google Developer Training

Lab: Scripting the Service Worker

This lab walks you through creating a simple service worker.

Contents:

Overview

What you will learn

- How to create a basic service worker script, install it, and do simple debugging

What you should know

- Basic JavaScript and HTML
- Familiarity with the concept and basic syntax of ES2015 Promises
- The concept of an Immediately Invoked Function Expression (IIFE)
- How to clone GitHub repos and checkout branches from the command line
- How to enable the developer console

What you need before you begin

- Computer with terminal/shell access
- Connection to the internet
- A browser that supports service workers:
    - Chrome 55+

    - Firefox 50+
    - Opera 38+
    - Android Browser 51+
    - Chrome for Android 51+
- A text editor

1. Setting Up

Download and unzip the repository, **service-worker-lab.zip**. Change to the base directory with the following command:

```
$ cd service-worker-lab
```

This repository has multiple files:

- *README.md* is documentation for GitHub, and can be ignored
- Sample resources that we use in testing:
    - *other.html*
    - *other.css*
    - *other.js*
    - *another.html*
    - *another.css*
    - *another.js*
- *index.html* is the main HTML page for our sample site/application
- *index.css* is the stylesheet for *index.html*
- *service-worker.js* is the JavaScript file that is used to create our service worker

2. Start a local web server and load the page

Check out the starting code branch:

```
$ git checkout 02_setting-up
```

Start a local web server in the project directory. Then, open your browser and navigate to the appropriate local host port (for example, *localhost:8000/*). Use incognito browsing in this lab unless stated otherwise. It ensures that your program is starting from a known state.

3. Registering the Service Worker

Service workers must be registered.

Open *service-worker.js*. Note that the file contains only an empty function. We have not implemented any code to run within the service worker yet.

Open *index.html* in your text editor, find the `<script>` tag, and replace the TODO comment with the following code:

```
if (!('serviceWorker' in navigator)) {
  console.log('Service worker not supported');
  return;
}
navigator.serviceWorker.register('service-worker.js')
.then(function() {
  console.log('Registered');
})
.catch(function(error) {
  console.log('Registration failed:', error);
});
```

Save the script and refresh the page. The console should return a message indicating that the service worker was registered.

Optional: open the site on an unsupported browser and verify that the support check conditional works.

Explanation

Always begin by checking whether the browser supports service workers. The service worker is exposed on the window's Navigator object and can be accessed with `window.navigator.serviceWorker`. In this case, if service workers aren't supported, the script logs a message and fails immediately.

Calling `serviceWorker.register(...)` "registers" the service worker, installing the service worker's script. This returns a promise that resolves once the service worker is successfully registered. If the registration fails, so will the promise.

4. Listening for Lifecycle Events

The service worker uses events to report its status to the application.

Open *service-worker.js* in your text editor. Replace the first TODO comment with the following code:

```
self.addEventListener('install', function(event) {
  console.log('Service worker installing...');
});

self.addEventListener('activate', function(event) {
  console.log('Service worker activating...');
});
```

Close and re-open the page to activate our new service worker. The console log should now show registration, installation, and activation events.

Note that all pages associated with the service worker must be closed before an updated service worker can take over.
Note that registration may appear out of order. The service worker runs concurrently with the page, so we can't guarantee the order of the logs. Installation, activation, and other service worker events occur in a defined order inside the service worker and should always appear in the expected order.

Explanation

This service worker emits an 'install' event at the end of registration. In this case we log a message, but this is a good place for caching static assets.

When a service worker is registered, the browser detects if the service worker is new (either because it is different from the previously installed service worker or because it's the first service worker for this site). If the service worker is new (as it is in this case) then the browser installs it.

The service worker also sends an 'activate' event when it takes control of a page. We log a message here, but this event is often used to update a cache.

Only one service worker can be active at a time, so a new service worker isn't activated until the existing service worker is no longer in use. This is why all pages controlled by a service worker must be closed before a new service worker can take over. Since we closed and reopened the page, the new service worker was activated.

Note that you can manually activate a new service worker in some browser's developer tools, overriding the default behavior.
Optional: Change the service worker by adding a comment. Save the file and refresh the page. Notice that the new service worker installs but does not activate. This is because the old service worker still controls the page.

Note that browsers sometimes will not detect changes to a service worker instantaneously. If you are getting unexpected results, wait a minute or so in between page loads.
Registering an existing (not new) service worker has no effect; the existing service worker remains installed and active.

Optional: Close and reopen the page. Reload the page a second time. Notice how the sequence of events changes. After initial installation and activation, re-registering an existing worker does not re-install or re-activate the service worker.

Note in the previous experiment that the service worker was installed and activated on the first load, even though it was not new. This is because service workers do not actually persist in incognito mode.
Optional: Navigate to the appropriate local host port using standard browsing mode (not incognito mode). Note that installation and activation occur normally. Close and reopen the page (still in standard browsing mode). Observe this time that installation and activation did not occur because the existing service worker persisted.

For more information

Intro to Service Worker

Check the status of service workers in developer tools

Using Service Workers - MDN

Service Workers Explained

5. Intercepting Network Requests

Service Workers can act as a proxy between your web app and the browser.

Replace the final TODO in *service-worker.js* with:

```
self.addEventListener('fetch', function(event) {
  console.log('Fetching:', event.request.url);
});
```

Close and reopen the page (returning to incognito mode) to install and activate the updated service worker. The console log should indicate that the new service worker was registered and, installed, and activated.

Note that it is possible for fetch events to be logged even if they occurred before activation. This is because the service worker runs concurrently with the page and fetch is

asynchronous, so activation may complete before fetch. See Intro to Service Worker for more info.
Now reload a second time. Notice that you see fetch events in the console for each of the page's assets (javascript, css, etc.)

Click the link to "Other page", "Another page", and "Back" to see the differences in the fetch event URL's in the console log. Do all the logs make sense?

Note that if you visit a page multiple times and do not have the cache disabled, CSS and JavaScript assets may be cached locally. If this occurs you will not see fetch events for these resources.

Explanation

The service worker sends a fetch event for every HTTP request made by the browser. The fetch event object contains the fetch request. Listening for fetch events in the service worker is similar to listening to click events in the DOM. In this case when a fetch event occurs, we log the requested URL to the console. In practice we could also create our own custom response with arbitrary resources.

For more information

Fetch Event - MDN

Using Fetch - MDN

Introduction to Fetch - Google Developer

Solution code

To get a copy of the working code, use the following commands:

```
$ git reset --hard
$ git checkout 05_basic-service-worker
```

6. Optional: Explore a Service's Worker's "Scope"

Service workers have scope. The scope of the service worker determines from which paths the service worker intercepts requests.

Update the registration code in *index.html* with:

```
if (!('serviceWorker' in navigator)) {
  console.log('Service worker not supported');
  return;
}
navigator.serviceWorker.register('service-worker.js')
.then(function(registration) {
  console.log('Registered at scope:', registration.scope);
})
.catch(function(error) {
  console.log('Registration failed:', error);
});
```

Refresh the browser. Notice that the console shows the scope of the service worker (for example *http://localhost:8000/*).

Explanation

The promise returned by `register()` resolves to the registration object, which contains the service worker's scope.

The default scope is the path to the service worker file, and extends to all lower directories. This service worker controls requests from all files at this domain because *service-worker.js* is located in the root directory.

Optional: Move the service worker into the *below* directory and update the service worker URL in the registration code. Close and reopen the page. The console shows that the scope of the service worker is now *http://localhost:8000/below/* (port number may vary). Click the link to "Other page", "Another page" and "Back". The console is now only logging fetch events for *another.html, another.css, and another.js*, because these are the only resource within the service worker's scope.

It is possible to set an arbitrary scope by passing in an additional parameter when registering, for example:

```
navigator.serviceWorker.register('/service-worker.js', {
  scope: '/app/'
});
```

In the above example the scope of the service worker is set to `/app/`. The service worker intercepts requests from pages in `/app/` and `/app/lower/` but not from pages like `/app` or `/`.

Note that you cannot set an arbitrary scope that is above the service worker's actual

location.
Optional: Move the service worker back out into the project root directory and update the service worker URL in the registration code (if you had changed these in the previous Optional section). Set the scope of the service worker to the *below* directory using the optional parameter in `register()`. Close and reopen the page. Again the console shows that the scope of the service worker is now *localhost:8000/below/* (port number may vary) and logs fetch events only for *another.html, another.css, and another.js*. Click the links to “Other page”, “Another page” and “Back”, to confirm.

For more information

Service worker registration object

The register() method

Service worker scope

Solution code

To get a copy of the working code, use the following commands:

```
$ git reset --hard
$ git checkout 06_scoped-service-worker
```

7. Congratulations

You now have a simple service worker up and running. You can learn more about service workers and what they can do in Intro to Service Worker.

Lab: Auditing with Lighthouse

Lighthouse is an open-source tool from Google that audits a web app for PWA features. It provides a set of metrics to help guide you in building a PWA with a full application-like experience for your users.

Contents:

Overview

What you will learn

- How to use Lighthouse to audit your Progressive Web Apps

What you should know

- Basic JavaScript and HTML
- How to clone GitHub repos and checkout branches from the command line

What you need before you begin

- Connection to the internet
- Chrome browser

- A text editor

1. Setting up

Download and unzip the starter repository, **lighthouse-lab.zip**. Change to the current working directory with the following command:

```
$ cd lighthouse-lab
```

This repository has multiple files:

- *README.md* is documentation for GitHub, and can be ignored
- The images directory has images for the app and home screen icon
- *main.css* is the CSS for the application
- *index.html* is the main HTML page for the application

2. Starting a local web server

Check out the starting code branch:

```
$ git checkout 02_setting-up
```

Start a local web server in the project directory. Then, open your browser and navigate to the appropriate local host port (for example, *http://localhost:8000/*).

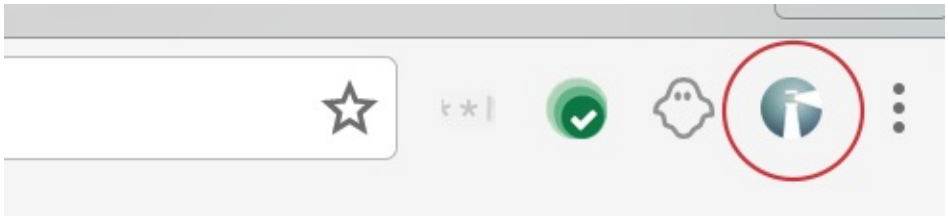
You can see that we have a responsive blog site.

3. Installing Lighthouse

Lighthouse is available as a Chrome extension for Chrome 52 and later.

Download the Lighthouse Chrome extension from the Chrome Web Store.

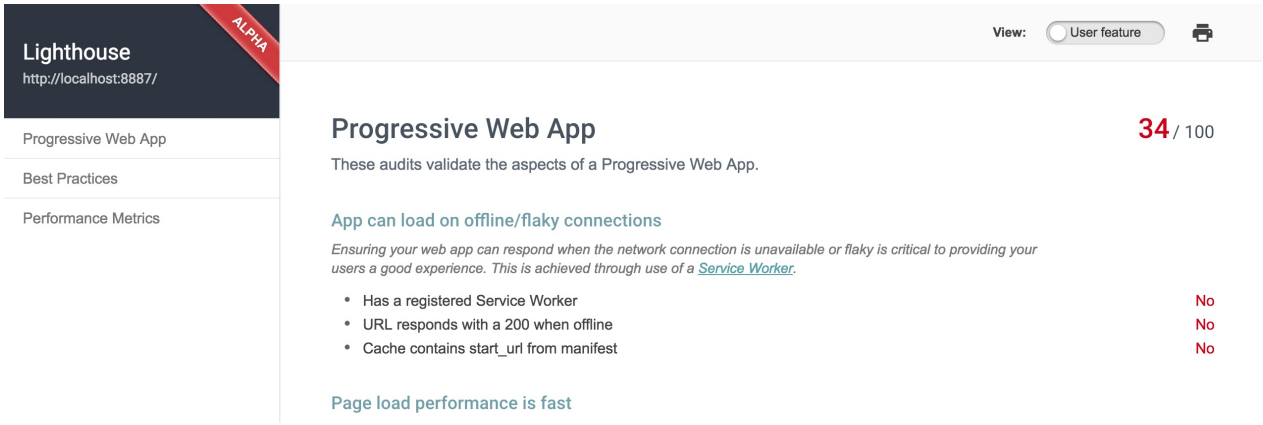
When installed it'll place an icon in your taskbar.



4. Testing the app

Navigate to the localhost page where the app is hosted. Click on the Lighthouse icon and choose to generate a report.

Lighthouse will run the report and generate an HTML page with the results. The report page should look similar to this:



Looks like we have a pretty low score. Take a moment to look through the report and see what is missing.

5. Adding a service worker

We can see from the report that having a service worker is necessary.

5a Registering a service worker

Create an empty JavaScript file in the root directory and name it *service-worker.js*. This is going to be our service worker file. Now replace TODO 5a in *index.html* with the following:

```
<script>
  (function() {
    if (!('serviceWorker' in navigator)) {
      console.log('Service worker not supported');
      return;
    }
    navigator.serviceWorker.register('service-worker.js')
    .then(function(registration) {
      console.log('SW successfully registered');
    })
    .catch(function(error) {
      console.log('registration failed', error);
    });
  })();
</script>
```

5b Caching offline & start pages

The report also indicates that our app must respond with a 200 when offline and must have our starting URL (“start_url”) cached.

Add the following code to the empty *service-worker.js* file:

```
self.addEventListener('install', function(event) {
  event.waitUntil(
      caches.open('static-cache-v1')
        .then(function(cache) {
          return cache.addAll([
            'index.html',
            'css/main.css'
          ]);
        })
    );
});

self.addEventListener('fetch', function(event) {
  event.respondWith(caches.match(event.request)
  .then(function(response) {
        if (response) {
          return response;
        }
        return fetch(event.request);
      })
    .catch(function() {
      console.log('resource not available');
    })
  );
});
```

Refresh the page (for the app, not the Lighthouse page). Check the console and check that the service worker has registered successfully.

Explanation

We have created a service worker for our app and registered it. Don't worry if you don't understand the code inside the service worker. Here is what it does:

1. The first block (install event listener) caches the files *index.html* and *main.css* so that they are saved locally. This allows us to access them even when offline, which is what the next block does.
2. The second block (fetch event listener) intercepts requests for resources and checks first if they are cached locally. If they are, the browser gets them from the cache and doesn't need to make a network request.

This allows us to respond with a 200 even when offline. Once we have loaded the app initially, both the files needed to run the app (*index.html* & *main.css*) are saved in the cache. If the page is loaded again, the browser grabs the files from the cache regardless of network conditions. This also lets us satisfy the requirement of having our starting URL (*index.html*) cached.

6. Adding a manifest file

The report also indicates that we need a manifest file.

6a. Creating the manifest file

Create an empty file called *manifest.json* in the root directory. Then replace TODO 6a in *index.html* with the following:

```
<!-- Web Application Manifest -->
<link rel="manifest" href="manifest.json">
```

6b. Adding manifest code

Add the following to the *manifest.json* file:

```
{
  "name": "Demo Blog Application",
  "short_name": "Blog",
  "start_url": "index.html",
  "icons": [{
        "src": "images/touch/icon-72x72.png",
        "sizes": "72x72",
        "type": "image/png"
      }, {
        "src": "images/touch/icon-96x96.png",
        "sizes": "96x96",
        "type": "image/png"
      }, {
        "src": "images/touch/icon-128x128.png",
        "sizes": "128x128",
        "type": "image/png"
      }, {
        "src": "images/touch/ms-touch-icon-144x144-precomposed.png",
        "sizes": "144x144",
        "type": "image/png"
      }, {
        "src": "images/touch/apple-touch-icon.png",
        "sizes": "152x152",
        "type": "image/png"
      }, {
        "src": "images/touch/chrome-touch-icon-192x192.png",
        "sizes": "192x192",
        "type": "image/png"
      },{
        "src": "images/touch/chrome-splashscreen-icon-384x384.png",
        "sizes": "384x384",
        "type": "image/png"
      },{
        "src": "images/touch/icon-512x512.png",
        "sizes": "512x512",
        "type": "image/png"
      }],
  "background_color": "#3E4EB8",
  "display": "standalone",
  "theme_color": "#2E3AA1"
}
```

6c. Adding tags for other browsers

Replace TODO 6c in *index.html* with the following:

```
<!-- Chrome for Android theme color -->
<meta name="theme-color" content="#2E3AA1">

<!-- Tile color for Win8 -->
<meta name="msapplication-TileColor" content="#3372DF">

<!-- Add to homescreen for Chrome on Android -->
<meta name="mobile-web-app-capable" content="yes">
<meta name="application-name" content="PSK">
<link rel="icon" sizes="192x192" href="images/touch/chrome-touch-icon-192x192.png">

<!-- Add to homescreen for Safari on iOS -->
<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="Polymer Starter Kit">
<link rel="apple-touch-icon" href="images/touch/apple-touch-icon.png">

<!-- Tile icon for Win8 (144x144) -->
<meta name="msapplication-TileImage" content="images/touch/ms-touch-icon-144x144-preco
mposed.png">
```

Explanation

We have created a manifest file and “add to homescreen” tags. Don’t worry about the details of the manifest and these tags. Here is how they work:

1. Chrome uses *manifest.json* to know how to style and format some of the progressive parts of your app, such as the “add to homescreen” icon and splash screen.
2. Other browsers don’t (currently) use the *manifest.json* file to do this, and instead rely on HTML tags for this information. While Lighthouse doesn’t require these tags, we’ve added them because they are important for supporting as many browsers as possible.

This allows us to satisfy the manifest related requirements of Lighthouse (and a PWA).

For more information

Add to home screen

Web app manifests

Solution code

To get a copy of the working code, use the following commands:

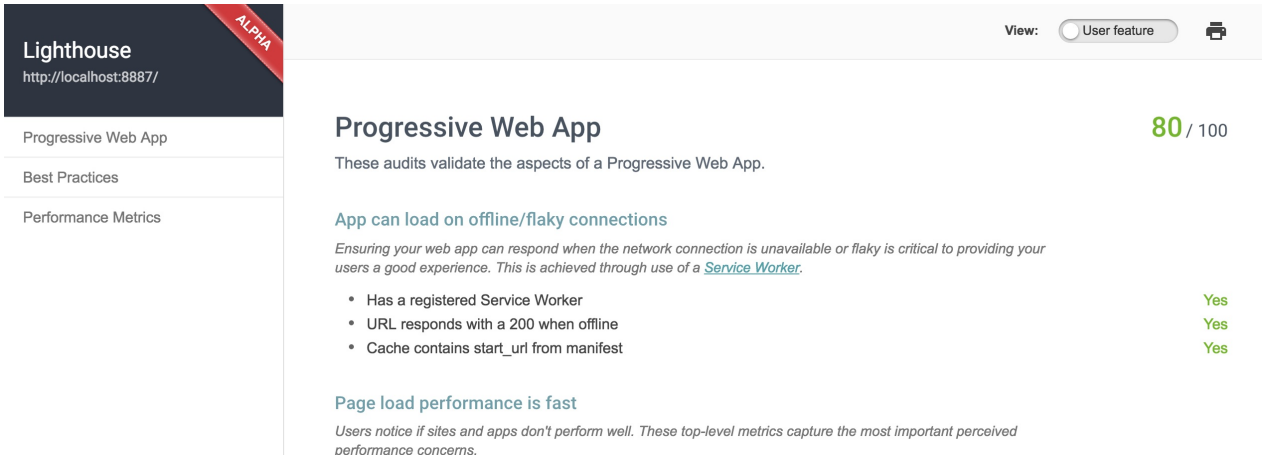
```
$ git reset --hard
$ git checkout 06_solution
```

7. Testing the updated app

Now we need to retest the app to see our changes. Return to the localhost page where that app is hosted. Click the Lighthouse icon and choose to generate a report (you may be prompted to close dev tools if they are still open).

Now we should have passed many more tests.

You may need to clear the cache to see the improved results. Go to **chrome://history/** and click **Clear browsing data** and clear the data from the past hour. Now refresh the app and click the the Lighthouse icon.
The report should look something like this:



Now our score is much better.

You can see that we are still missing the HTTPS requirements, since we are using a local server. In production, service workers require HTTPS, so you'll need to use that.

8. Optional: Lighthouse from the Command Line

If you want to run Lighthouse from the command line (for example, to integrate it with a build system) it is available as a Node module that you can install from the terminal.

If you haven't already, download Node and select the version that best suits your environment and operating system.

Install Lighthouse's Node module from the terminal:

```
$ npm install -g GoogleChrome/lighthouse
```

Configure Chrome to run Lighthouse.

```
$ npm explore -g lighthouse -- npm run chrome
```

The command looks in your *node_modules* directory for Lighthouse and then calls an NPM script that launches Chrome with special flags Lighthouse needs to run its tests.
Run Lighthouse on a demo Progressive Web App (or the app you just made):

```
$ lighthouse https://airhorner.com/
```

You can check Lighthouse flags and options with:

```
$ lighthouse --help
```

9. Congratulations

You now know how to use the Lighthouse tool to audit your Progressive Web Apps.

Offline support for an existing site

Contents:

1. Introduction

This workshop will provide an overview of offline web applications, their rationale and how to add offline capabilities to your application using service workers.

What you will learn

- How to add offline capabilities to an application

What you should know

- Semantic HTML and CSS
- Javascript and ES6 Promises

2. Why use offline

Most of the next billion users to come online will do so via mobile devices as their primary or only means of access. Cellular data can be prohibitively expensive and Wifi data can be unreliable giving rise to lie-fi (your device shows it's connected but in reality it's not).

Until the introduction of service workers we couldn’t make our applications work offline in any meaningful way. Previous attempts at working offline (application cache) were partially successful at the cost of implicit behaviors that were not always easier to understand.

Using Service Workers we gain control over users' offline experiences in our applications. We can tailor our experience to work offline with cached data and have the data refresh when the user goes online.

3. Set up

We will work in the `starting-code` folder. If at any point you need help in understanding the final result you can open the `solution-code` folder to see what the final code looks like

Download and unzip the repository, **offline-quickstart-lab.zip**. Change to the current working directory with the following command:

```
$ cd starting-code
```

4. Start a local web server and load the page

Start a local web server at starting-code/app. Then, open your browser and navigate to the appropriate local host port (for example, *http://localhost:8000/*).

5. How to do offline

To enable offline capabilities in an application you need to do two things:

- Create the service worker
- Add the code to cache resources
- Add code to fetch resources from the cache
- Add code to your index.html page to register the service worker

Code in service-worker.js

Open `service-worker.js` and paste the following code:

```
self.addEventListener('fetch', function(event) {
  event.respondWith(caches.match(event.request)
  .then(function(response) {
        if (response) {
          return response;
        }
        return fetch(event.request);
      })
    .catch(function() {
      console.log('resource not available');
    })
  );
});
```

Make sure you save the file before moving on.

Explanation

The code to add files to the cache is already in `service-worker.js`.

This example of a `fetch` event handler does 3 things:

- Tries to match the request with the content of the cache
- If not successful then it attempts to get the resource from the network using the Fetch API
- If the cache doesn't have the resource and the network request fails (possibly because the user is offline) then we catch the error and log a message to the console

Code in index.html

Open index.html. Towards the end of the file there is a `script` tag with a comment inside (insert your service worker related code here).

Paste the following code inside the script tag. It's ok to delete the comment while doing so.

```
    if ('serviceWorker' in navigator) {
      navigator.serviceWorker.register('service-worker.js')
        .then(function(registration) {
          console.log('Service Worker registration successful with scope: ' +
           registration.scope);
        })
        .catch(function(err) {
          console.log('Service Worker registration failed: ', err);
        });
    }
```

Save the file and refresh the page in the browser. Stop the server (Ctrl+C if your server is in the command window) to simulate going offline, and refresh the page. The page should load normally!

Explanation

This script performs the following tasks:

- Checks if the user's browser supports service workers by testing if the `serviceWorker` property exist in the navigator object
- If the check succeeds we register our service-worker.js script
- Log success to console along with the scope the service worker covers
- If the registration fails log a message to console along with the reason for the error

Promises Codelab

Promises are a new way to handle asynchronous code in Javascript. Promises have been around for a while in the form of libraries, such as:

- Q
- when
- WinJS
- RSVP.js

The promise libraries listed above and promises that are part of the ES2015 JavaScript specification (also referred to as ES6) are all Promises/A+ compatible.

Contents:

Introduction

This codelab uses promises to teach you about the following tasks.

What you'll learn

- How to use promises to build a promise chain to asynchronous code
- How to run multiple promises simultaneously
- How to store data for use offline later
- Create timers to run against promises-based functions

What you'll need

- Chrome 52 or above
- Web Server for Chrome, or your own web server of choice
- The sample code (available from Github)
- A text editor
- Basic knowledge of HTML, CSS, JavaScript, and Chrome DevTools

This codelab is focused on using promises with Progressive Web Apps. Non relevant concepts and code blocks are glossed over and are provided for you to simply copy and paste as a way to expedite the exercises.

Getting set up

Download the Code

Download and unzip the repository, **promises-lab.zip**.

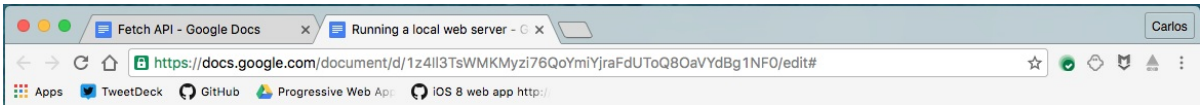
Web Server for Chrome

Web Server for Chrome is a Chrome extension that creates a local web server. It runs on all platforms where Chrome runs including Chromebooks.

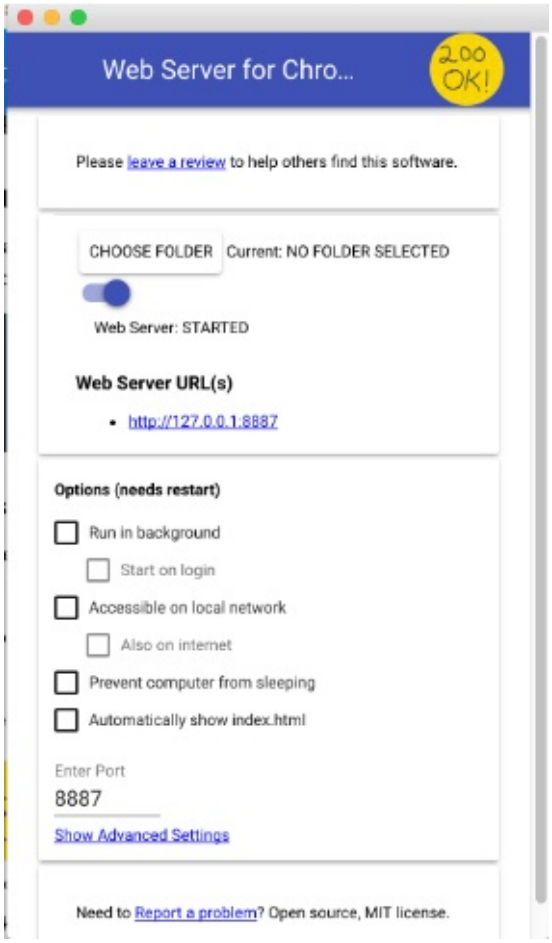
Download Web Server for Chrome from the Chrome Store.

To open the web server extension:

1. Make sure your Bookmark bar is open. Go to the View menu and ensure that **Always Show Bookmark Bar** is checked.
2. Click on the **Apps** button (far left in the image below)



3. Select the Web Server Icon Web Server When the web server la*f*unches you will

see the following window: 

To run your code click ***Choose Folder*** and point to the folder hosting your code and click the URL under **Web Server URL(s)**. Check **Automatically show index.html** to test a web app.

For other options see: **Running a local web server.**

Create an Account on geonames.org

Before we get started you must create an account on **geonames.org**.

1. Go to the following URL: geonames.org/login
2. Register an account. You will use the account for all the exercises below.

Build a Promise-Based Function

Repository: https://github.com/caraya/promises-lab

Folder: starting-code

Files: exercise02.js and exercise02.html

Run the following in the command line to check out the code:

$ git checkout https://github.com/caraya/promises-lab

Open the **js/codelab02.js** file and copy its contents.

Open the **exercise02.html** file and paste the contents of **codelab02.js** into the empty script tag at the bottom of the file.

Testing

Open **exercise02.html**. The Spanish flag should appear on screen.

Explanation

This exercise builds a function that performs the following tasks:

- Retrieve data from geonames.org.
- Extract the country name for the JSON received.
- Fetch the flag for the country.
- Append the flag to the DOM to display on the browser.

The `promiseFlag` function runs a promise chain to retrieve the flag of the specified country.

```
function promiseFlag(country) {
  getCountryInfo(country)
  .then(getCountryName)
  .then(fetchFlag)
  .then(displayFlag)
  .catch(function(err) {
    console.log(err);
  });
}
```

To gain flexibility and avoid long anonymous functions in the promise chain, you can create functions to handle each phase of the process to retrieve a flag.

```
function getCountryInfo(country) {
  var apiCall = 'http://api.geonames.org/searchJSON?q=' + country +
  '&maxRows=1' + '&username=caraya';

  return fetch(apiCall)
  .then(response => {
    if (response.status != 200) {
      throw new Error('Unable to fetch file');
    }
    console.log(response);
    return response.json();
  })
  .catch((err) => {
    console.log('Fetch failed: ' + err);
  });
}
```

The first function: `getCountryInfo` builds a query URL to geonames.org using the country name, a fixed value for `maxRows` and a fixed value for the user name of the user making the request.

The function then fetches the URL and throws an error if the response code is anything but 200, meaning that the requested completed successfully.

If the request completed successfully, then return the response as a JSON object.

```
function getCountryName(json) {
  var country = json.geonames[0].countryName;
  return country;
}
```

`getCountryName` parses the JSON object returned from the last function to obtain the country name. Then, it returns the name of the country parsed from the JSON object.

```
function fetchFlag(country) {
  var url = 'https://caraya.github.io/promises-lab/end-code/flags/';
  var countryFlag = url + country + '.png';
  return fetch(countryFlag, {mode: 'cors'})
  .then(response => {
    if (!response.ok) {
      throw new Error('Fetch failed' + response.status);
    }
    return response.blob();
  });
}
```

The `fetchFlag` function uses the country name to generate a URL that fetches the flag of the corresponding country from a Github pages site. Next, we check if the response succeeded by testing for theresponse.ok value in the response object.

If the response succeeds, then return the response object parsed as a blob.

```
function displayFlag(flagBlob) {
  var flagImage = document.createElement('img'); // 1
  var flagDataURL = URL.createObjectURL(flagBlob); // 2
  flagImage.src = flagDataURL; // 3
  document.body.appendChild(flagImage); //4
}
```

The final function, `displayFlag` takes the blob containing the flag and performs the following steps:

1. Create an image element (flagImage).
2. Create an objectURL for the blob (flagDataURL).
3. Attach the objectURL to the image source attribute.
4. Append the image to the image element to the body of the document.

   ```
   A blob cannot be attached directly to an HTML element. You must convert the blob t
   o a data URL (as show in displayFlag) that can be attached to HTML elements such a
   s the image's source attribute.
   ```

Promise.race

File: js/codelab03.js and exercise03.html

Open the **js/codelab03.js** file and copy the contents. Open the **exercise03.html** file and paste the script in the empty script tag at the bottom of the file.

Testing

Open exercise03.html in your web browser. The Spanish flag should load.

If it doesn't open your Dev Tools (Control + Shift + I or F12 in Windows. Command + Shift + I on Macintosh) and check if the timeout promise rejected (the console would show **Timeout Triggered**).

Explanation

When using Promise.race we match two functions against each other. A common use of the promise.race function is to test a a function against a timer (see the delay function in the previous example). If the function being tested resolves first then Promise.race will resolve. We can set the timeout function to reject after a certain time, so if the timeout function executes first, then the function being tested is not executed and Promise.race is rejected.

This exercise uses three functions: delay, new Promise, promise.race.

- The delay function takes one parameter, ms, representing the number of milliseconds before the setTimeout triggers and the promise resolves. In the body of the function. we return a new promise with a timeout of ms milliseconds.

```
function delay(ms) {
return new Promise(function(resolve, reject) {
setTimeout(reject, ms);
});
}
```

- The `promiseFlag` function (described in the previous exercise) retrieves a flag.
- The third function, promise.race, is shown in the following code and creates a promise race between fetching a resource in an external server and a delay of 2500 milliseconds using the delay function we just described. If the delay function resolves then we throw an error and stop processing the function.

```
function timedLoad() {
Promise.race([
promiseFlag('Spain'),
delay(2500)
])
.then(function() {
console.log('image loaded first');
})
.catch(function(reason) {
console.log('Function took more than 2500 miliseconds');
});
}
```

  If promiseFlag finishes first, then the Spanish flag displays. If the delay function executes first, then it means that the race took longer than 2500 milliseconds and the timeout triggered.

Promise.all

Using Promise.all we can group two or more promises to run at the same time and only resolve when all promises have resolved and reject if any of the promises reject (the remaining promises being discarded).

Repository: https://github.com/caraya/promises-lab

Folder: starting-code

File: js/codelab04.js and exercise04.html

Open the **js/codelab04.js** file and copy the contents. Open the **exercise04.html** file and paste the script in the empty script tag at the bottom of the file.

Explanation

The following code reuses the `promiseFlag` function to retrieve flags. This time it creates constants to hold our separate calls to `promiseFlag` for the flags of Spain, Argentina, and Armenia.

```
const image1 = promiseFlag('Spain');
const image2 = promiseFlag('Argentina');
const image3 = promiseFlag('Armenia');
```

We call Promise.all and pass it an array of all the promises we want to work with in parallel.

```
Promise.all([image1, image2, image3])
.then(function() {
  console.log('All images loaded successfully');
})
.catch(function(message) {
  console.log('One or more images failed to load ' + message);
});
```

When all the promises resolve successfully then move to the then statement and return a success message to the console.

If any of the `promiseFlag` promises rejects, then the entire Promise.all will reject and log the failure message to the console, along with the reason for the rejection.

Note that because the calls to `promiseFlag` are asynchronous you are not guaranteed to get the flags in the same order every time, only that you will get the same flags.

Lab: Fetch API

This tutorial will walk you through using the Fetch API. The Fetch API is a simple interface for fetching resources and an alternative to the XMLHttpRequest API.

Contents:

Overview

What you will learn

- How to use the Fetch API to request resources
- How to make GET, HEAD, and POST requests with fetch

What you should know

- Basic JavaScript and HTML

- Familiarity with the concept and basic syntax of ES2015 Promises
- The concept of an Immediately Invoked Function Expression (IIFE)
- How to clone GitHub repos and checkout branches from the command line
- How to enable the developer console
- Some familiarity with JSON

What you will need

- Computer with terminal/shell access
- Connection to the internet
- A browser that supports fetch
  - Edge 14+
  - Firefox 47+
  - Chrome 49+
  - Opera 38+
  - Android browser 51+
  - Chrome for Android 51+
- A text editor

The Fetch API is not currently supported in all browsers, and there is a polyfill (see Appendix for installation instructions).

1. Setting up

Download and unzip the repository, **fetch-api-lab.zip**. Change to the right directory with the following command:

```
$ cd fetch-api-tutorial
```

This repository contains multiple files:

- *README.md* is documentation for GitHub, and can be ignored
- Sample resources (image, JSON, and text) that we use in testing
  - *kitten.jpg*
  - *example.txt*
  - *example.json*
- *index.html* is the main HTML page for our sample site/application
- *package.json* is used to install the fetch polyfill if needed
- *main.js* is the main JavaScript file that we write our code in
- binaries for running an echo server

2. Start a local web server and load the page

Checkout the starting code branch:

```
$ git checkout 02_setting-up
```

Start a local web server in the project directory. Then, open your browser and navigate to the appropriate local host port (for example, *http://localhost:8000/*).

3. Fetching JSON

To support browsers that don't implement fetch, always start with a support detection conditional. Open *main.js* in your text editor and replace the TODO with the following code:

```
if (!('fetch' in window)) {
  console.log('Fetch API not found, try including the polyfill');
  return;
}
// We can safely use fetch from now on

fetch('/examples/example.json')
.then(function(responseAsJSON) {
  console.log(responseAsJSON);
})
.catch(function(error) {
  console.log('Fetch failed', error);
});
```

Save the script and refresh the page. The console should log the fetch response.

Optional: open the site on an unsupported browser and verify that the support check conditional works.

Explanation

We pass the absolute path for the resource we want to retrieve as a parameter to fetch. A promise that resolves to a Response object is returned.

Response objects represent the response to a request. They contain the response body and also useful properties and methods.

3.1 Response properties

Find the values of response.ok, response.status, and response.type for */examples/example.txt* and */examples/non-existent.txt*. The values should be different for the two files (do you understand why?). If you got any console errors when you logged these values, do the values match up with the context of the error?

Explanation

Why didn't a failed response activate the `catch()` block? This is an important note for fetch and promises — bad responses (like 404's) still resolve! A fetch promise only rejects if the request was unable to complete. You must always check the validity of the response.

3.2 Checking response validity

Update the previous code to to check the validity of responses:

```
if (!('fetch' in window)) {
  console.log('Fetch API not found, try including the polyfill');
  return;
}
// We can safely use fetch from now on

fetch('/examples/example.json')
.then(function(responseAsJSON) {
  if (!responseAsJSON.ok) {
    throw Error(responseAsJSON.statusText);
  }
  console.log(responseAsJSON);
})
.catch(function(error) {
  console.log('Fetch failed', error);
});
```

Now bad responses throw an error and `catch()` takes over. Save the script. Try fetching both */examples/example.txt* and */examples/non-existent.txt*. Note the differences in the console logs now.

For more information

Response objects

3.3 Abstracting fetch with functions

We are going abstract the contents of the `then()` and `catch()` blocks into functions. This keeps the code better organized.

Replace the previous code with the following:

```
if (!('fetch' in window)) {
  console.log('Fetch API not found, try including the polyfill');
  return;
}
// We can safely use fetch from now on
```

```
function readResponseAsJSON(responseAsJSON) {
  if (!responseAsJSON.ok) {
    throw Error(responseAsJSON.statusText);
  }
  console.log(responseAsJSON);
}

function logError(error) {
  console.log('Fetch failed', error);
}

fetch('/examples/example.json')
.then(readResponseAsJSON)
.catch(logError);
```

Save the script and refresh the page. Confirm that the new code functions the same as the original code.

4. Reading the response

Responses must be read in order to access the body of the response. Response objects have methods for doing this.

Replace the `readResponse()` function with the following:

```
function readResponseAsJSON(responseAsJSON) {
  if (!responseAsJSON.ok) {
    throw Error(responseAsJSON.statusText);
  }
  return responseAsJSON.json();
}
```

Now create another function that we can use to log the result. Add this code to *main.js*:

```
function logResult(result) {
  console.log(result);
}
```

Update the fetch chain to the following:

```
fetch('/examples/example.json') // Step 1
.then(readResponseAsJSON) // Step 2
.then(logResult) // Step 3
.catch(logError);
```

(This is promise chaining.)

Save the script and refresh the page. Check the console to see that JSON is being logged.

Explanation

Step 1. Fetch is called on a resource, example.json. Fetch returns a promise that will resolve to a Response object. When the promise resolves, the response object is passed to `readResponseAsJSON()`.

Step 2. `readResponseAsJSON()` checks the validity of the response. (This is particularly important. Without this check bad responses are passed down the chain. That could break later code that may rely on receiving a valid response.) Next, the response object is read with `response.json()`. Response.json() reads the response and returns a promise that resolves to JSON. The JSON object will be passed to `logResult()`.

Step 3. Finally, the JSON object from the original request to example.json is logged.

If a promise rejects or an error is thrown at some point, then `catch()` takes control.

For more information

Response.json()

Response methods

Promise chaining

Solution code

To get a copy of the working code, use the following commands:

```
$ git reset --hard
$ git checkout 04_json-example
```

5. Fetching images

Fetch is not limited to JSON. In this example we will fetch an image and append it to the page.

Uncomment the empty image tag in *index.html:*

```
<img src="" alt="Image placeholder">
```

Add the following functions to *main.js*:

```
function readResponseAsBlob(response) {
  if (!response.ok) {
    throw Error(response.statusText);
  }
  return response.blob();
}

function showImage(responseAsBlob) {
  const myImage = document.querySelector('img');
  const myImageUrl = URL.createObjectURL(responseAsBlob);
  myImage.src = myImageUrl;
}
```

Update the fetch chain to the following:

```
fetch('/images/kitten.jpg')
.then(readResponseAsBlob)
.then(showImage)
.catch(logError);
```

Save the script and refresh the page. You should see an image of a kitten.

Explanation

In this example *images/kitten.jpg* is fetched. Response.blob() reads the response into a Blob. The URL object's `createObjectURL()` method is used to generate a data URL representing the Blob. That data URL is set as the `src` attribute in the image tag.

For more information

Blobs

Response.blob()

URL object

Solution code

To get a copy of the working code, use the following commands:

```
$ git reset --hard
$ git checkout 05_img-example
```

6. Fetching text

In this example we will fetch text and add it to the page.

Add the following functions to *main.js*:

```
function updatePage(responseAsText) {
  var container = document.querySelector('#container');
  container.textContent = responseAsText;
}

function readResponseAsText(response) {
  if (!response.ok) {
    throw Error(response.statusText);
  }
  return response.text();
}
```

Update the fetch chain to the following:

```
fetch('/examples/example.txt')
.then(readResponseAsText)
.then(updatePage)
.catch(logError);
```

Save the script and refresh the page. You should see new content on the page.

Explanation

Here, *examples/example.txt* is fetched. Response.text() reads the response as text. This text is then used to set the content of the `<body>` of the page.

For more information

Response.text()

Solution code

To get a copy of the working code, use the following commands:

```
$ git reset --hard
$ git checkout 06_text-example
```

Note that the methods used in the previous examples are actually methods of Body, a Fetch API mixin that is implemented in the Response object.

7. HEAD requests

By default, fetch uses the GET method, which retrieves a specific resource. But fetch can also use other methods.

Update the fetch chain to the following:

```
fetch('/examples/example.txt', {
  method: 'HEAD'
})
.then(readResponseAsText)
.then(updatePage)
.catch(logError);
```

Save the script and refresh the page. What do you notice about the page's content now?

Explanation

`fetch()` can receive a second optional parameter, `init`. This allows the creation of custom settings for the request, such as the request method, cache mode, credentials, and more.

Here we have set the method to HEAD using the `init` parameter. HEAD requests are just like GET requests, except the body of the response is empty. This kind of request can be used when all you want is metadata about a file but don't need to transport all of the file's

data. For example, the HEAD method could be used to request the size of a resource without actually loading the resource itself. That information could then determine how or if to request the full resource.

Now does it make sense that the page is is empty?

For more information

HTTP methods

Fetch method signature

Solution code

To get a copy of the working code, use the following commands:

```
$ git reset --hard
$ git checkout 07_head-example
```

8. POST requests

Fetch can send data with POST requests.

Update the fetch chain to the following:

```
fetch('http://localhost:5000/', {
  method: 'POST',
  body: 'username=john&password=12345',
})
.then(readResponseAsText)
.then(logResult)
.catch(logError);
```

Save the script. If you are using OS X (mac), run:

```
$ ./echo-server/echo-server_darwin_amd64
```

If you are using Windows, run:

```
> echo-server\echo-server_windows_amd64
```

If you are using Linux, run:

```
$ ./echo-server/echo-server_linux_amd64
```

Refresh the page. Do you see the sent request echoed in the console? Does it contain the username and password?

Explanation

To make a POST request with fetch, we again use the `init` parameter to specify the method. This is also where we set the body of the request, which is the data to send.

The terminal/shell command sets up a simple server at http://localhost:5000/ that echoes back the requests sent to it. When data is sent as a POST request to http://localhost:5000/, the request is returned, read as text, and logged to the console.

Note that you can stop the server by pressing the ctrl key and c key simultaneously.
In practice this server would be a 3rd party API.

For more information

Go language

Solution code

To get a copy of the working code, use the following commands:

```
$ git reset --hard
$ git checkout 08_post-example
```

9. Optional: Use multiple methods

Consider the scenario in which a requested resource is unavailable. Rather than break the user experience, fetch can be used to request an alternative resource and deliver that instead.

Request *non-existent.json*. If the file is available (indicated by a successful response code) your code should read the response as JSON, and log the JSON in the console. If the file is not available, your code should instead request *example.txt*, read the response as text, and update the page `<body>` with the resulting text.

Test the code with both *example.json* and *non-existent.json* to make sure it works for existent and non-existent files.

Solution code

To get a copy of the working code, use the following commands:

```
$ git reset --hard
$ git checkout 09_multiple-example
```

Optional: Restore your code to the image fetching example. Try requesting an image from another domain, such as unsplash. Note that the fetch fails. This is because fetch supports Cross Origin Resource Sharing (CORS), and can only request resources from servers with the appropriate headers.

10. Congratulations

You now know how to use the Fetch API to request resources and post data to servers.

11. Additional resources

- MDN Documentation

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

https://developer.mozilla.org/en-US/docs/Web/API/GlobalFetch/fetch

- Google Developer Intro

https://developers.google.com/web/updates/2015/03/introduction-to-fetch?hl=en

- Blog posts

https://davidwalsh.name/fetch

https://jakearchibald.com/2015/thats-so-fetch/

12. Appendix

Installing the Fetch Polyfill

If you have node installed, you can add the fetch polyfill by executing

```
$ npm install
```

from within the project directory. Node's package manager, npm, will read the dependencies described in the provided *package.json* file and install the fetch polyfill module in your project. To actually incorporate the polyfill into our code, we need to reference it. Do this by uncommenting the appropriate `<script>` tag in the head of *index.html*, which should look like

```
<script src="/node_modules/whatwg-fetch/fetch.js"></script>
```

Lab: Caching files with the service worker

This lab covers the basics of caching files with the service worker. The technologies involved are the Cache API and the Service Worker API. See the Caching files with the service worker doc for a full tutorial on the Cache API. See Introduction to Service Worker and Lab: Scripting the service worker for more information on service workers.

Contents:

Overview

What you will learn

- How to use the Cache API to access and manipulate data in the cache
- How to cache the application shell and offline pages
- How to intercept network requests and respond with resources in the cache

What you should know

- Basic JavaScript and HTML
- Familiarity with the concept and basic syntax of ES2015 Promises
- How to clone GitHub repos and checkout branches from the command line

- Familiarity with DevTools in Chrome

What you will need

- Computer with terminal/shell access
- Connection to the internet
- A text editor

1. Setting up

Download and unzip the repository, **cache-api-lab.zip**. Change to the current working directory with the following command:

```
$ cd cache-api-lab
$ git checkout 02-setting-up
```

This repository contains:

- An images folder containing some sample images
- A pages folder containing some sample pages and a custom offline page
- A scripts folder containing a cache polyfill
- A style folder containing the app's CSS
- *index.html* is the home page for our sample site/application
- *service-worker.js* is the service worker where we will code the interactions with the cache

2. Start a local web server and load the page

Start a local web server in the project directory. Then, open your browser and navigate to the appropriate local host port (for example, *http://localhost:8000/*).

3. Caching the application shell

Cache the application shell in the "install" event handler in the service worker.

Replace TODO 1 in serviceworker.js with the following code:

service-worker.js

```
var filesToCache = [
  '/',
  'style/main.css',
  'index.html',
  'pages/offline.html'
];

var staticCacheName = 'pages-cache-v1';

self.addEventListener('install', function(event) {
  console.log('Attempting to install service worker and cache static assets');
  event.waitUntil(
    caches.open(staticCacheName)
    .then(function(cache) {
      return cache.addAll(filesToCache);
    })
    .catch(function(error) {
      console.log('Installation failed, ', error);
    })
  );
});
```

Test the code in the browser. Refresh the page once or twice for the browser to recognize the new service worker, then hold Shift and click the refresh icon again to update the service worker. Open DevTools by right-clicking on the page and selecting “Inspect”. In DevTools, go to the “Application” tab and expand “Cache storage”. You should see the files appear in the table on the right. You may need to refresh the page again for the changes to appear.

Explanation

We first define the files to cache and assign them the to the `filesToCache` variable. These files make up the ‘application shell’: the static HTML and CSS files that give your app a unified look and feel. We also assign a cache name to a variable, so that updating the cache name (and by extension the cache version) happens in one place.

In the install event handler we create the cache with caches.open and use the addAll method to add the files to the cache. We wrap this in event.waitUntil to extend the lifetime of the event until all of the files are added to the cache and addAll resolves successfully.

For more information

Intro to PWA Architecture

The install event - MDN

4. Serving files from the cache

Now that we have the files cached, we can intercept requests for those files from the network and respond with the files from the cache.

Replace TODO 2 in service-worker.js with the following:

service-worker.js

```
self.addEventListener('fetch', function(event) {
  console.log('Fetch event for ', event.request.url);
  event.respondWith(
    caches.match(event.request).then(function(response) {
      if (response) {
        console.log('Found ', event.request.url, ' in cache');
        return response;
      }
      console.log('Network request for ', event.request.url);
      return fetch(event.request)

      // TODO 3 - Add fetched files to the cache

      ;
    }).catch(function(error) {

    // TODO 4 - Respond with custom offline page

    })
  );
});
```

In the browser, Shift+Reload the page to update the service worker. Refresh once more to see the log messages appear in the console. Let's test out the app offline. In DevTools, go to the "Network" tab and click on the dropdown menu that says "No throttling". Choose "Offline (0ms, 0kb/s, 0kb/s)" from the list and refresh the page. The page should load normally!

Explanation

The "fetch" event listener intercepts all requests. We use event.respondWith to create a custom response to the request. Here we are using the "cache falling back to network" strategy: we first check the cache for the requested resource and then, if that fails, we send the request to the network.

For more information

caches.match - MDN

The Fetch API

The fetch event - MDN

5. Add network responses to the cache

We can add files to the cache as they are requested.

Replace TODO 3 in the fetch event handler with the code to add the files returned from the fetch to the cache:

service-worker.js

```
.then(function(response) {
  return caches.open(staticCacheName).then(function(cache) {
    cache.put(event.request.url, response.clone());
    return response;
  });
});
```

In DevTools, change the network throttling back to “No throttling”. Shift+Reload the page to update the service worker. Visit at least one of the links on the home page. Change the network throttling to “Offline (0ms, 0kb/s, 0kb/s)”. Now if you revisit the pages they should load normally! Try navigating to some pages you haven’t visited before.

We need to pass a clone of the response to cache.put, because the response can only be read once. See Jake Archibald’s What happens when you read a response for an explanation.

For more information

Cache.put - MDN

6. Respond to network errors with offline page

Replace TODO 4 with the code to respond with the offline.html page from the cache. The catch will trigger if the fetch to the network fails.

Change the network throttling back to “No throttling”. Shift+Reload to update the service worker. Navigate to a page you haven’t visited before to see the custom offline page.

Solution code

```
$ git reset --hard
$ git checkout 06-offline-page
```

7. Deleting outdated caches

We can get rid of unused caches in the service worker “activate” event.

Replace TODO 5 in service-worker.js with the following code:

service-worker.js

```
self.addEventListener('activate', function(event) {
  console.log('Activating new service worker...');

  var cacheWhitelist = [staticCacheName];

  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.map(function(cacheName) {
          if (cacheWhitelist.indexOf(cacheName) === -1) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

Try changing the name of the cache to `pages-cache-v2` :

service-worker.js

```
var staticCacheName = 'pages-cache-v2';
```

Test the code in the browser. Shift+Reload the page to activate the new service worker. In DevTools, go to the Application tab and expand “Cache storage”. You should see just the new cache. The old cache `pages-cache-v1` has been removed.

Explanation

We delete old caches in the “activate” event to make sure that we aren’t deleting caches before the service worker has taken over the page. We create an array of caches that are currently in use and delete all other caches.

For more information

Promise.all - MDN

Array.map - MDN

Solution code

```
$ git reset --hard
$ git checkout 07-delete-caches
```

8. Congratulations

You have learned the basics of using the Cache API in the service worker. We have covered caching the application shell, intercepting network requests and responding with items from the cache, adding resources to the cache as they are requested, responding to network errors with a custom offline page, and deleting unused caches.

9. More resources

Cache API

https://developer.mozilla.org/en-US/docs/Web/API/Cache

Using service workers

https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers

The Offline Cookbook

https://jakearchibald.com/2014/offline-cookbook/

Lab: IndexedDB

This codelab guides you through the basics of using the IndexedDB API. This lab uses Jake Archibald's IndexedDB Promised library, which is very similar to the IndexedDB API, but uses promises rather than events. This simplifies the API while maintaining its structure, so anything you learn using this library can be applied to the IndexedDB API directly.

Contents:

1. Overview

This lab builds a furniture store app, *Couches-N-Things*, to demonstrate the basics of IndexedDB.

What you will learn

- How to create object stores and indexes
- How to create, retrieve, update, and delete values (a/k/a CRUD)
- How to use cursors
- (Optional) How to use the `getAll()` method

What you should know

- Basic JavaScript and HTML
- JavaScript Promises
- How to clone GitHub repos and check out branches from the command line

What you will need

- Computer with terminal/shell access
- Chrome

2. Setting Up

Download and unzip the repository, **indexed-db-lab.zip**. Change to the current working directory with the following command:

```
$ cd indexed-db-lab
$ git checkout 02-setting-up
```

This repo contains:

- **index.html** - contains some forms for testing our IndexedDB database
- **js/main.js** - where we will write the scripts to interact with the database
- **js/idb.js** - the IndexedDB Promised library

3. Check for support

Because IndexedDB isn't universally supported by all browsers, we must check for support before using it.

Replace TODO 1 in main.js with the following code:

main.js

```
if (!('indexedDB' in window)) {return;}
```

4. Create the database and add items

4.1 Opening a database

Create the database for your app.

In main.js, replace TODO 2 with the following code:

main.js

```
var dbPromise = idb.open('couches-n-things', 1);
```

Open index.html in Chrome and open DevTools. For Mac users, go to the View menu and open **Developer** > **DevTools**. Alternatively, right-click on the page and click “Inspect”. In DevTools, go to the “Application” tab and expand IndexedDB to see your new database.

4.2 Deleting a database

If at any point in the codelab your database gets into a bad state, you can delete it from the console with the following command:

console

```
indexedDB.deleteDatabase('couches-n-things')
```

Go to your app in Chrome and enter the above command into the console. You should see a message display in the console confirming that the database has been deleted.

Do not refresh the page. We are going to re-create the database from scratch in the next step.

4.3 Create an object store

Create an object store in the database to hold the furniture objects.

Comment the idb.open line in main.js:

main.js

```
// var dbPromise = idb.open('couches-n-things', 1);
```

Replace TODO 3 in main.js with the following:

main.js

```
var dbPromise = idb.open('couches-n-things', 1, function(upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
      console.log('Creating the products object store');
      upgradeDb.createObjectStore('products', {keyPath: 'id'});

    // TODO 5 - create ‘name’ index

    // TODO 6 - create ‘price’ and ‘description’ indexes

    // TODO 10 - create an ‘orders’ object store

  }
});
```

Reload the page in the browser. Go to **DevTools** > **Application** again and expand the ‘couches-n-things’ database in IndexedDB. You should see the empty ‘products’ object store.

Explanation

Object stores are created inside the callback function in idb.open, which executes only if the version number is greater than the existing version in the browser or if the database doesn’t exist. The callback is passed the UpgradeDB object, which is used to create the object stores.

Inside the callback, we include a switch block that executes its cases based on the version of the database already existing in the browser. Case 0 executes if the database doesn’t yet exist (make sure to delete database before executing this code).

We have specified the `id` property as the `keyPath` for the object store. Objects added to this store must have an `id` property and the value must be unique.

We are deliberately not including break statements in the switch block to ensure all of the cases after the starting case will execute.

For more information

createObjectStore method

4.4 Add objects to the object store

Add some sample furniture items to the object store.

Replace TODO 4 in main.js with the following code:

main.js

```
dbPromise.then(function(db) {
  var tx = db.transaction('products', 'readwrite');
  var store = tx.objectStore('products');
  var items = [
    {
      name: 'Couch',
      id: 'cch-blk-ma',
      price: '499.99',
      color: 'black',
      material: 'mahogany',
      description: 'A very comfy couch',
      quantity: 3
    },
    {
      name: 'Armchair',
      id: 'ac-gr-pin',
      price: '299.99',
      color: 'grey',
      material: 'pine',
      description: 'A plush recliner armchair',
      quantity: 7
    },
    {
      name: 'Stool',
      id: 'st-re-pin',
      price: '59.99',
      color: 'red',
      material: 'pine',
      description: 'A light, high-stool',
      quantity: 3
    },
    {
      name: 'Chair',
      id: 'ch-blu-pin',
      price: '49.99',
      color: 'blue',
      material: 'pine',
      description: 'A plain chair for the kitchen table',
      quantity: 1
    },
    {
      name: 'Dresser',
      id: 'dr-wht-ply',
      price: '399.99',
      color: 'white',
      material: 'plywood',
      description: 'A plain dresser with five drawers',
      quantity: 4
    },
    {
```

```
      name: 'Cabinet',
      id: 'ca-brn-ma',
      price: '799.99',
      color: 'brown',
      material: 'mahogany',
      description: 'An intricately-designed, antique cabinet',
      quantity: 11
    }
  ];
  items.forEach(function(item) {
    console.log('Adding item: ', item);
    store.add(item);
  });
  return tx.complete;
}).then(function() {
  console.log('All items added successfully!');
}).catch(function(e) {
  console.log('Error adding items: ', e);
});
```

Save the file and reload the page in the browser. Click **Add Products** and refresh the page. Confirm that the objects display in the `products` object store in the DevTools view of your browser.

Explanation

All database operations must happen within a transaction. The transaction rolls back any changes to the database if any of the operations fail. This ensures the database is not left in a partially updated state.

Specify the transaction mode as ‘readwrite’ when making changes to the database (i.e. using the add or put methods).

For more information

Transactions - MDN

Add method - MDN

Solution code

```
$ git reset --hard
$ git checkout 04-4-populate-db
```

5. Searching the database

5.1 Creating indexes

Create some indexes on your object store.

Replace TODO 5 in main.js with the following code:

main.js

```
case 1:
  console.log('Creating a name index');
  var store = upgradeDb.transaction.objectStore('products');
  store.createIndex('name', 'name', {unique: true});
```

Remember to change the version number to 2 before you test the code in the browser. The full idb.open method should look like this:

main.js

```
var dbPromise = idb.open('couches-n-things', 2, function(upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
      console.log('Creating the products object store');
      upgradeDb.createObjectStore('products', {keyPath: 'id'});
    case 1:
      console.log('Creating a name index');
      var store = upgradeDb.transaction.objectStore('products');
      store.createIndex('name', 'name', {unique: true});

    // TODO 6 - create ‘price’ and ‘description’ indexes

    // TODO 10 - create an ‘orders’ object store

  }
});
```

Save the file and reload the page in the browser. Confirm that the indexes display in the `products` object store in the DevTools view of your browser.

Explanation

In the example, we create an index on the `name` property, allowing us to search and retrieve objects from the store by their name. The optional `unique` option ensures that no two items added to the 'products' object store use the same name.

For more information

IDBIndex - MDN

createIndex method - MDN

5.2 Create ‘price’ and ‘description’ indexes

Replace TODO 6 in the switch block with a case 2 to add ‘price’ and ‘description’ indexes to the object store. Do not include the optional `{unique: true}` argument. Remember to change the version number of the database to 3 before testing the code.

Solution code

```
$ git reset --hard
$ git checkout 05-2-create-indexes
```

5.3 Using the get method

Use the indexes you created in the previous section to retrieve items from the store.

Replace TODO 7 in main.js with the following code:

main.js

```
var key = document.getElementById('name').value;
if (key === '') {return;}
var s = '';
dbPromise.then(function(db) {
  var tx = db.transaction('products', 'readonly');
  var store = tx.objectStore('products');
  var index = store.index('name');
  return index.get(key);
}).then(function(object) {
  if (!object) {return;}

  s += '<h2>' + object.name + '</h2><p>';
  for (var field in object) {
    s += field + ' = ' + object[field] + '<br/>';
  }
  s += '</p>';

}).then(function() {
  if (s === '') {s = '<p>No results.</p>';}
  document.getElementById('results').innerHTML = s;
});
```

Test the code in the browser. Enter an item name into the **By Name** field and click **Search** next to the text box. A furniture item should display on the page.

Explanation

This code calls the get method on the 'name' index to retrieve an item by its 'name' property. Then it displays the object and its properties on the page.

For more information

Get method - MDN

5.4 Using cursors

Use a cursor object to get items from your store within a price range.

Replace TODO 8 in main.js with the following code:

main.js

```
  var lower = document.getElementById('priceLower').value;
  var upper = document.getElementById('priceUpper').value;
  if (lower == '' && upper == '') {return;}

  var range;
  if (lower != '' && upper != '') {
    range = IDBKeyRange.bound(lower, upper);
  } else if (lower == '') {
    range = IDBKeyRange.upperBound(upper);
  } else {
    range = IDBKeyRange.lowerBound(lower);
  }
  var s = '';
  dbPromise.then(function(db) {
    var tx = db.transaction('products', 'readonly');
    var store = tx.objectStore('products');
    var index = store.index('price');
    return index.openCursor(range);
  }).then(function showRange(cursor) {
    if (!cursor) {return;}
    console.log('Cursored at:', cursor.value.name);

    s += '<h2>Price - ' + cursor.value.price + '</h2><p>';
    for (var field in cursor.value) {
      s += field + '=' + cursor.value[field] + '<br/>';
    }
    s += '</p>';

    return cursor.continue().then(showRange);
  }).then(function() {
    if (s === '') {s = '<p>No results.</p>';}
    document.getElementById('results').innerHTML = s;
  });
```

Test the code in the browser. Enter some prices into the 'price' text boxes (without a currency symbol) and click **Search**. Items should appear on the page ordered by price. Try doing the same test with an item description.

Optional: Replace TODO 9 in the getByDesc() function with the code to get the items by their descriptions. The first part is done for you. The function uses the 'only' method on IDBKeyrange to match all items with exactly the provided description.

Explanation

After getting the price values from the page, we determine which method to call on `IDBKeyRange` to limit the cursor. We open the cursor on the 'price' index and pass the cursor object to the `showRange` function in `.then`. This function adds the current object to the html string, moves on to the next object with `cursor.continue()`, and calls itself, passing in the

cursor object. `showRange` loops through each object in the object store until it reaches the end of the range. Then the cursor object is `undefined` and `if (!cursor) {return;}` breaks the loop.

For more information

IDBCursor - MDN

IDBKeyRange - MDN

cursor.continue() - MDN

Solution code

```
$ git reset --hard
$ git checkout 05-4-get-data
```

6. Process orders (optional)

6.1 Create an ‘orders’ object store

Create an object store to hold pending orders.

Replace TODO 10 in main.js with a case 3 that adds an ‘orders’ object store to the database. Make the keyPath the ‘id’ property. This is very similar to the ‘products’ object store creation in case 0. Remember to change the version number of the database to 4 so the callback executes.
Test the code in the browser. Confirm that the object store displays in DevTools.

6.2 Add sample orders

Replace TODO 11 in main.js with code to add the following items to the ‘orders’ object store:

main.js

```
var items = [
        {
          name: 'Cabinet',
          id: 'ca-brn-ma',
          price: '799.99',
          color: 'brown',
          material: 'mahogany',
          description: 'An intricately-designed, antique cabinet',
          quantity: 7
        },
        {
          name: 'Armchair',
          id: 'ac-gr-pin',
          price: '299.99',
          color: 'grey',
          material: 'pine',
          description: 'A plush recliner armchair',
          quantity: 3
        },
        {
          name: 'Couch',
          id: 'cch-blk-ma',
          price: '499.99',
          color: 'black',
          material: 'mahogany',
          description: 'A very comfy couch',
          quantity: 3
        }
      ]
```

Test the code in the browser. Confirm that the objects show up in the ‘orders’ store in the DevTools view of your browser.

Solution code

```
$ git reset --hard
$ git checkout 06-2-add-orders
```

6.3 Display orders

Replace TODO 12 in main.js with the code to display all of the objects in the ‘orders’ object store on the page. This is very similar to the `getByPrice` function except you don’t need to define a range for the cursor. The code to insert the ‘s’ variable into the html is already written.

Solution code

```
$ git reset --hard
$ git checkout 06-3-display-orders
```

6.4 Get all orders

Replace TODO 13 in the `fulfillOrders` function in main.js with the code to get all objects from the 'orders' object store. You must use the getAll() method on the object store. This returns an array containing all the objects in the store, which is then passed to the `processOrders` function in .then.

Solution code

```
$ git reset --hard
$ git checkout 06-4-get-orders
```

6.5 Process the orders

This step processes the array of orders passed to the `processOrders` function.

Replace TODO 14 in main.js with the following code:

main.js

```
return dbPromise.then(function(db) {
  var tx = db.transaction('products');
  var store = tx.objectStore('products');
  return Promise.all(
    orders.map(function(order) {
      return store.get(order.id).then(function(product) {
        return decrementQuantity(product, order);
      });
    })
  );
});
```

Explanation

This code gets each object from the 'products' object store with an id matching the corresponding order, and passes it and the order to the `decrementQuantity` function.

For more information

Promise.all() - MDN

Array.map() - MDN

6.6 Decrement quantity

Now we need to check if there are enough items left in the 'products' object store to fulfill the order.

Replace TODO 15 in main.js with the following code:

main.js

```
return new Promise(function(resolve, reject) {
  var item = product;
  var qtyRemaining = item.quantity - order.quantity;
  if (qtyRemaining < 0) {
    console.log('Not enough ' + product.id + ' left in stock!');
    document.getElementById('receipt').innerHTML =
    '<h3>Not enough ' + product.id + ' left in stock!</h3>';
    return reject(new Error('Out of stock!'));
  }
  item.quantity = qtyRemaining;
  return resolve(item);
});
```

Explanation

Here we are subtracting the quantity ordered from the quantity left in the 'products' store. If this value is less than zero, we reject the promise. This causes `Promise.all` in the `processOrders` function to fail so that the whole order is not processed. If the quantity remaining is not less than zero, then update the quantity and return the object.

For more information

new Promise - MDN

6.7 Update the ‘products’ store

Finally, we must update the ‘products’ object store with the new quantities of each item.

Replace TODO 16 in main.js with the code to update the items in the ‘products’ objects store with their new quantities. We already updated the values in the `decrementQuantity` function and passed the array of updated objects into the `updateProductsStore` function. All that’s left to do is use [ObjectStore.put](https://developer.mozilla.org/en-US/docs/Web/API/IDBObjectStore/put) to update each item in the store. A few hints:

- Remember to make the transaction mode ‘readwrite’
- Remember to return transaction.complete (tx.complete) after putting each item into the store

Solution code

```
$ git reset --hard
$ git checkout 06-7-update-products
```

7. Congratulations

You have learned the basics of working with IndexedDB. You have learned the commands to create, read, update and delete data in the database. Finally, you have worked with the `getAll` command and used cursors to iterate over the data.

Project: Building a social media app

Tooling and automation codelab

In this session we will discuss tools and automation to make your development workflow easier. We will concentrate on the Gulp task runner and tasks that will make creating a service worker and testing for performance easier. Because of scope and time constraints we will not discuss how to build general purpose workflows but provide a good starting point.

Content:

Why an automated build system?

Modern web development is full or repetitive tasks like concatenating and minimizing CSS and JavaScript, compressing images, converting SASS to CSS and many others. Gulp and related tools automate these tasks making it easier to configure and run these repetitive tasks so developers no longer need to complete them manually.

What is Gulp?

Gulp is a streaming task runner. It provides basic tooling and infrastructure for you to build your projects. You can read more in the Tooling and Automation chapter, but you don't have to be a Gulp expert for this project.

For more information about Gulp check the Gulp Getting Started guide. To get an idea of what Gulp can do check the list of Gulp recipes on Github

Download and unzip the repository, **gulp-lab.zip**. Change to the current working directory with the following command:

```
$ cd gulp-lab
```

The starting code is in the **starting-code** directory.

The code in the starting-code branch provides the basic structure of a web application. We'll use this structure as the starting point for the exercises below:

1. Basic Gulpfile

1. Install the project dependencies running `npm install`

1. Open `gulpfile.js`. At the bottom of the script below the comment "insert your code here" past the task below

```
  gulp.task('service-worker', function(callback) {
    swPrecache.write(path.join(paths.src, 'service-worker.js'), {
      staticFileGlobs: [
        paths.src + '*.html',
        paths.src + 'js/**/*.js',
        paths.src + 'css/*.css',
        paths.src + 'apple-touch-icon.png',
        paths.src + 'img/**/*.{svg,png,jpg,gif}'
      ],
      importScripts: [
        './js/sw-toolbox.js',
        './js/toolbox-scripts.js'
      ],
      stripPrefix: paths.src
    }, callback);

  });
```

This code will create a service worker at the root of your application

1. Open app/index.html at the bottom of the file there is an empty script tag with a comment: "paste your service worker code here". Inside that tag paste the code below.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('service-worker.js')
    .then(function(registration) {
      console.log('Service Worker registration successful with scope: ' +
       registration.scope);
    })
    .catch(function(err) {
      console.log('Service Worker registration failed: ', err);
    });
}
```

2. A more complex task

In this exercise we will implement a task that adds prefixes and sourcemaps for your CSS files.

1. From your terminal run the following command to install the plugins for the exercise:

- gulp-sourcemaps
- gulp-autoprefixer

  Hint: remember to use npm install --save-dev

Once the plugins are installed open gulpfile.js. At the top of the file require the plugins you just installed. The variable names should be sourcemaps (for gulp-sourcemaps) and autoprefixer (for gulp-autoprefixer)

Paste the task below to the Gulpfile. This is the task that will take the CSS files, add prefixes to the css based on data from caniuse.com and write them to the css directory

```
gulp.task('processCSS', function() {
 return gulp.src('./css/**/*.css')
  .pipe(sourcemaps.init())
  .pipe(autoprefixer())
  .pipe(sourcemaps.write())
  .pipe(gulp.dest('./css'));
});
```

3. Serving files and watching for changes

Typing the same command over and over can get tedious and very error prone. Gulp can help automate tasks using watches. For this exercise follow these steps:

1. Install browser-sync
2. Browser sync uses a special form of require. Use `var browserSync = require('browser-sync').create();`
3. Copy the code below into your Gulpfile.

```
gulp.task('serve', ['processCSS'], function() {

    browserSync.init({
        server: "./app"
    });

    gulp.watch("app/css/**/*.css", ['processCSS']);
});

gulp.task('default', ['serve']);
```

In this step we explore some additional capabilities available in Gulp.

- We create a task that depends on another. Serve will not run before processCSS has completed. It will then serve the contents from the app directory.
- We create a watcher that will run whenever a CSS file changes> It will run processCSS to make sure our prefixes are correct
- We also set a default task that will run whenever we run gulp with no parameters. When this happens Gulp will execute the serve task

Lab: Integrating Web Push

Push Notifications combine messages that originate from the server and locally-triggered notifications, and work even when the user is not actively using your application. The notification system is built on top of the Service Worker API, which receives push messages in the background and relays them to your application.

Contents:

1. Overview

What you will learn

- How to create and display a notification in a web application, with and without user-required actions
- How to use the Web Push API to receive notifications, adapting to different browser implementations
- How to implement regular data updates in the background
- How to design push notifications into your app following best practices

What you should know before you begin

- Basic understanding of Chrome developer tools.
- Basics of the Service Worker API
- Basics of JavaScript Promises.

- Intermediate experience using the command-line interface.
- Intermediate-to-advanced experience with JavaScript.
- How to clone GitHub repos and check out branches from the command line

What you will need

- Computer with terminal/shell access
- Connection to the Internet
- A Google or Gmail account
- Chrome
- Firefox
- Node.js

2. Setting Up

Download and unzip the repository, **push-notification-lab.zip**. Change to the current working directory with the following command:

```
$ cd Push-Notification-Codelab
$ git checkout 02-setting-up
```

This repo contains:

- An images folder containing sample images
- **js/main.js** where we will write the app's script
- A samples folder containing sample landing pages
- A boilerplate **index.html** file
- The service worker, **sw.js**, where we will write the script to handle notifications

3. Start a local web server and load a page

Follow the instructions in Running a local web server to install and run the appropriate server for your setup.

Open your browser and navigate to the appropriate local host port (for example, *http://localhost:8000/*).

If you are using Python to serve the app, open another terminal because the server blocks the terminal. In the new terminal window, change to the **Push-Notification-Codelab** directory.

4. Notification API

Push Notifications are assembled using two APIs: the Notification API and the Push API. The Notification API allows us to display system notifications to the user.

4.1 Check for support

Because notifications are not yet fully supported by all browsers, we must check for support.

Replace TODO 1 in **main.js** with the following code:

main.js

```
if (!('Notification' in window)) {
  console.log('This browser does not support notifications!');
}
```

In a practical application we would perform some logic to compensate for lack of support, but for our purposes we can log an error.

4.2 Request permission

Replace TODO 2 in **main.js** with the following code:

main.js

```
Notification.requestPermission(function(status) {
  console.log('Notification permission status:', status);
});
```

Open the app in Chrome. If the prompt does not appear, then click on the icon to the left of the URL and change the notification permission to “Ask by default”. Try rejecting the prompt and then check the console. Now reload the page and this time allow notifications. You should see a permission status of “granted” in the console.

Explanation

This opens a popup when the user first lands on the page prompting them to allow or block notifications. Once the user accepts you can display any notification. This permission is stored along with your app, so calling this again returns the user's last choice.

4.3 Display the notification

Replace TODO 3 in **main.js** with this code:

main.js

```
function displayNotification() {
  if (Notification.permission == 'granted') {
    navigator.serviceWorker.getRegistration().then(function(reg) {

    // TODO 4 - Add ‘options’ object to configure the notification

      reg.showNotification('Hello world!');
    });
  }
}
```

Save the file and reload the page in the browser so it picks up the changes. Click **allow** on the permission pop-up if needed. Now if you click **Notify me!** you should see a notification appear!

For more information

showNotification method - MDN

4.4 Add notification options

The notification can do much more than just display a title.

Replace TODO 4 with an options object:

main.js

```
var options = {
  body: 'Woohoo! First notification!',
  icon: 'images/notification-flat.png',
  vibrate: [100, 50, 100],
  data: {
    dateOfArrival: Date.now(),
    primaryKey: 1
  },

  // TODO 5 - add actions to the notification

};
```

Be sure to add the options object to the second parameter of `showNotification`:

main.js

```
reg.showNotification('Hello world!', options);
```

Save the code and reload the page in the browser. Now click **Notify me!** in the browser to see the new additions to the notification.

Explanation

`showNotification` has an optional second parameter which takes an object containing various configuration options. See the reference on MDN for more information on each option.

Attaching data to the notification when you create it allows your app to get that data back at some point in the future. Because notifications are created and live asynchronously to the browser you will frequently want to inspect the notification object after the user interacts with it so you can work out what to do. In practice, we can use a 'key' (unique) property in the data to determine which notification was called.

4.5 Add notification actions

To create a notification with a set of custom actions, we can add an actions array inside our notification options object.

Replace TODO 5 in the options object inside **main.js** with the following code:

main.js

```
actions: [
  {action: 'explore', title: 'Explore this new world',
    icon: 'images/checkmark.png'},
  {action: 'close', title: 'I don't want any of this',
    icon: 'images/xmark.png'},
]
```

Test this out in the browser. The notification now has two new buttons to click. These don't do anything yet. In the next sections we write the code to handle notification events and actions.

Explanation

The actions array contains a set of action objects that define the buttons that we want to show to the user. Actions get an ID when they are defined so that we can tell them apart in the service worker. We can also specify the display text, and add an optional image.

Solution code

```
$ git reset --hard
$ git checkout 04-5-display-notifications
```

4.6 Handle the 'notificationclose' event

User interactions with the notification raise events in the service worker.

Replace the TODO 6 in **sw.js** with an event listener for the `notificationclose` event:

sw.js

```
self.addEventListener('notificationclose', function(e) {
  var notification = e.notification;
  var primaryKey = notification.data.primaryKey;

  console.log('Closed notification: ' + primaryKey);
});
```

Let's test what we have so far. Go to the browser and hold Shift and click **refresh** to register the new service worker.

You can make sure that the new service worker takes over by going to **DevTools** >

Application > **Service worker** and clicking **Unregister** on the current service worker and refreshing the page. You may need to refresh the page more than once for the new service worker to take over.
Now, in the page, click **Notifiy me!** and then close the notification. Check the console to see the log message appear when the notification closes.

Explanation

This code gets the notification object from the event and then gets the data from it. This data can be anything we like. In this case, we get the value of the `primaryKey` property.

4.7 Handle the 'notificationclick' event

The most important thing to do is handle when the user clicks the notification.

Replace the TODO 7 in **sw.js** with this:

sw.js

```
self.addEventListener('notificationclick', function(e) {

  // TODO 8 - change the code to open a custom page

  // TODO 9 - change the code to handle actions

  clients.openWindow('http://google.com');
});
```

Now when you Shift+reload the page, open a new notification and click it, you should land on the Google homepage. Again, to make sure the new service worker activates, you can unregister the current service worker as described in the note above.

Optional: Inside the listener for the `notificationclick` event, replace TODO 8 with the code to get the notification from the event object and assign it to a variable called 'notification'. Then get the `primaryKey` from the data in the notification and assign it to a `primaryKey` variable. Replace the URL in `clients.openWindow` with `'samples/page' + primaryKey + '.html'`. Finally, at the bottom of the listener, add a line to close the notification. Refer to the Methods section in the Notification article on MDN to see how to programmatically close the notification.

In the browser, Shift+Reload and test the code. Now when you click the notification it should take you to **page1.html** and the notification should close after it is clicked. Try changing the `primaryKey` in **main.js** to 2 and test it again.

4.8 Handle actions

Let's add some code to the service worker to handle the actions.

To complete TODO 9, change the `notificationclick` event listener to look like this:

sw.js

```
self.addEventListener('notificationclick', function(e) {
  var notification = e.notification;
  var primaryKey = notification.data.primaryKey;
  var action = e.action;

  if (action === 'close') {
    notification.close();
  } else {
    clients.openWindow('samples/page' + primaryKey + '.html');
    notification.close();
  }
});
```

Test the code in the browser. Update the service worker and click **Notify me!**. Try clicking the actions.

There is very good reason for handling actions inside the `notificationclick` listener: not every platform supports actions buttons, and not every platform displays all of your actions, so centralizing everything in our 'notificationclick' event allows us to provide a default experience that works everywhere.

Solution code

```
$ git reset --hard
$ git checkout 04-8-handle-events
```

5. Push API

The Push API allows the server to push messages to the service worker, even while the app is not active.

For more information

Push API - MDN

Using the Push API - MDN

5.1 Install Node.js and setup the server

If you don't already have Node.js installed, download the installer and install the current node.js version for your operating system.

Check out the code to set up the server:

```
$ git reset --hard
$ git checkout 05-1-setup-server
```

5.2 Handle the push event

If a browser that supports push messages receives one, it registers a `push` event in the service worker.

Inside **sw.js** replace TODO 1 with the code to handle push events:

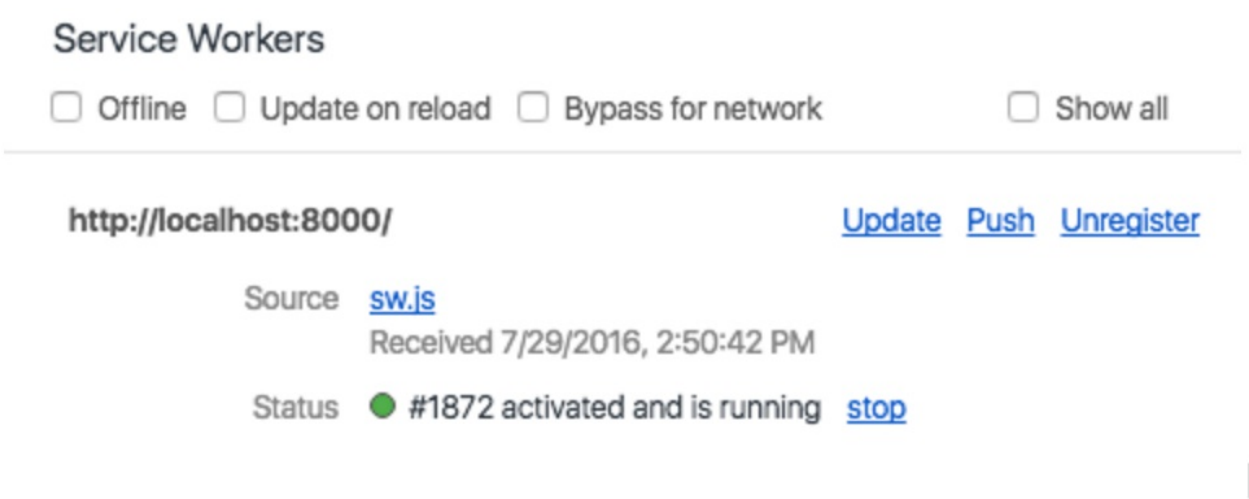
sw.js

```
self.addEventListener('push', function(e) {

  // TODO 4 - update push event handler to get data from the message

  var options = {
    body: 'This notification was generated from a push!',
    icon: 'images/notification-flat.png',
    vibrate: [100, 50, 100],
    data: {
      dateOfArrival: Date.now(),
      primaryKey: '-push-notification'
    },
    actions: [
      {action: 'explore', title: 'Explore this new world',
        icon: 'images/checkmark.png'},
      {action: 'close', title: 'I don’t want any of this',
        icon: 'images/xmark.png'},
    ]
  };
  e.waitUntil(
    self.registration.showNotification('Hello world!', options)
  );
});
```

Explanation

This event handler displays a notification similar to the ones we’ve seen before. The important thing to note is that the notification creation is wrapped in an `e.waitUntil` function. This extends the lifetime of the push event until the `showNotification` Promise resolves.

We can test this code using the `push` tool in Chrome DevTools, which allows us to manually trigger a push event from inside the browser. Save the code you’ve written so far and Shift+Reload the browser to install the new service worker. Open DevTools and go to the Service worker in the Resources tab. There should be a ‘Push’ link at the top of the service worker info. Click it to trigger the push event.



Be aware this UI is constantly changing, so it might look a little different when you try it.

For more information

Push Event - MDN

Solution code

```
$ git reset --hard
$ git checkout 05-2-push-event
```

5.3 Create a project on Firebase

To subscribe to the push service in Chrome, we need to create a project on Firebase.

1. In the Firebase console, select **Create New Project**.
2. Supply a project name and click **Create Project**.
3. Select the gear icon next to your project name at top left, and select **Project Settings**.
4. Select the **Cloud Messaging** tab. You can find your server key and sender ID in this page. Save these values.

Replace `YOUR SENDER ID` in the code below with the Sender ID of your project on Firebase and paste it into manifest.json (replace any code already there):

manifest.json

```
{
  "name": "Push Notifications codelab",
  "gcm_sender_id": "YOUR SENDER ID"
}
```

5.4 Check if the subscription object already exists

Check if a subscription object already exists for our app.

Replace the TODO 2 in **main.js** with the following code:

main.js

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.ready.then(function(reg) {
    reg.pushManager.getSubscription().then(function(sub) {
      if (sub == undefined) {
        // Update UI to ask user to register for Push
        console.log('Not subscribed to push service!');
      } else {
        // We have a subscription, update the database
        console.log('Subscription object: ', sub);
      }
    });
  });
}
```

If we don’t have subscription object, then we could add UI to the page prompting the user to enable push messages. Otherwise, we can send the latest subscription object to the database for storage.

For more information

PushManager - MDN

getSubscription method - MDN

5.5 Subscribe to the push service

Before sending any data via a push message, you must first subscribe to a push service.

Replace TODO 3 in **main.js** with the following code:

main.js

```
reg.pushManager.subscribe({
  userVisibleOnly: true
}).then(function(sub) {
  console.log('Endpoint URL: ', sub.endpoint);
}).catch(function(e) {
  if (Notification.permission === 'denied') {
    console.warn('Permission for notifications was denied');
  } else {
    console.error('Unable to subscribe to push', e);
  }
});
```

Let's try it out in the browser. Reload the service worker and check the console. You may need to refresh the page a couple times. If everything worked we should see an endpoint URL where we send the messages for that user display in the console, and the keys needed to encrypt the message payload. We use these in the next section to send a push message.

Explanation

Here we are subscribing to the `pushManager` on the registration object and logging the subscription object to the console. In production, it is your job to store this subscription object somewhere on your system.

The `.catch` handles the case in which the user has denied permission for notifications. In this case we might perform some logic to update our app to send messages to the user some other way.

Notice we are passing a flag named `userVisibleOnly` to the subscribe method. By setting this to `true`, we ensure that every incoming message has a matching notification. In the current implementation of Chrome, whenever we receive a push message and we don't have our site visible in the browser we must display a notification. If we don't display a notification the browser automatically creates one to let the user know that your web page is doing work in the background.

For more information

PushSubscription - MDN

Subscribe method - MDN

Notification permission status - MDN

Solution code

Caution: If you do check out the code, remember to replace the `gcm_sender_id` in **manifest.json** with your own Sender Id.

```
$ git reset --hard
$ git checkout 05-5-subscribe
```

5.6 Send your first Web Push Message using cURL

Let's use cURL to send a push message to Chrome through Firebase Cloud Messaging.

1. Make sure the latest service worker has control of the page by Shift-Reloading the page or unregistering the current service worker.
2. Click **Show Subscription Object** on the page and copy the last part of the endpoint (long alphanumeric string after the last slash) and paste it into the cURL command below.
3. Paste the Server Key you saved earlier into the Authorization header in the cURL command below.

   The Server Key can be found in your project on Firebase by clicking on the gear icon next to the project name and going to **Project settings** > **Cloud messaging**.
4. Paste the following cURL command (with your values substituted into the appropriate places) into a command window and hit enter:

```
curl --header "Authorization: key=SERVER KEY GOES HERE" --header "Content-Type: applic
ation/json" https://android.googleapis.com/gcm/send -d "{\"registration_ids\":[\"SUBSC
RIPTION ID GOES HERE\"]}"
```

Here is an example of what the cURL should look like with the correct info pasted in:

```
curl "https://android.googleapis.com/gcm/send" --request POST --header "Authorization:
 key=AIzaSyD1JcZ8WM1vTtH6Y0tXq_Pnuw4jgj_92yg" --header "Content-Type: application/json
" -d "{\"to\":\"cy041LQrB20:APA91bFTHyVEvgtK9xEuujt6LLkGvMVcJe7aUSn5b_B5EjSA6Mdj_Db7yS
dD8ZBGFw9Z1C5iZ1LRddIgkLjDr166ir3SLts65hKzyLstMP5NNwg6ZZikG7KjdB70O8fYcwC5OND8F6Yv\"}"
```

If the message sends successfully, then you should see something like this in the command window (it may take some time for the notification to display):

```
{"multicast_id":7007152374450936118,"success":1,"failure":0,"canonical_ids":0,"results
":[{"message_id":"0:1457450438558507%1fbab450f9fd7ecd"}]}
```

That's it! We have sent our very first push message. A notification should pop up on your screen. It may take a few seconds to appear.

Explanation

For FCM to push a notification to your web client, we need to send it a request that includes the following:

- The public Server key that you created earlier. FCM matches this with the Sender Id of the project in Firebase and in your `manifest.json`.
- A `Content-Type` header: `application/json`.
- An array of subscription IDs. The subscription ID is the last part of the subscription endpoint URL, after the last forward slash (`/`)

For more information

Push Demo

Getting Started with cURL

cURL Documentation

5.7 Get data from the push message

Chrome and Firefox support the ability to deliver data directly to your service worker via the push message.

To complete TODO 4, update the `push` event listener in **sw.js** to get the data from the message:

sw.js

```
self.addEventListener('push', function(e) {
  if (e.data) {
    var data = e.data.json();
    var title = data.title;
    var body = data.body;
    var primaryKey = data.primaryKey;
    console.log(data);
  } else {
    var title = 'Push message no payload';
    var body = 'Default body';
    var primaryKey = 1;
  }

  var options = {
    body: body,
    icon: 'images/notification-flat.png',
    vibrate: [100, 50, 100],
    data: {
      dateOfArrival: Date.now(),
      primaryKey: primaryKey
    },
    actions: [
      {action: 'explore', title: 'Explore this new world',
        icon: 'images/checkmark.png'},
      {action: 'close', title: 'I don’t want any of this',
        icon: 'images/xmark.png'},
    ]
  };
  e.waitUntil(
    self.registration.showNotification(title, options)
  );
});
```

In this example, we’re getting the data payload as JSON and extracting the title, body, and primaryKey.
We’ve now created everything necessary to handle the notifications in the client, but we have not yet sent the data from our server. That comes next.

For more information

Push Event data - MDN

5.8 Push the message

We can get all the information we need to send the push message to the right push service (and from there to the right client) from the subscription object.

Replace TODO 5 in **node/main.js** with the following code:

Be sure to replace the key in `webPush.setGCMAPIKey` with your own Server Key from your project on Firebase.

node/main.js

```
var webPush = require('web-push');

// TODO 6 - generate VAPID keys

var subscriptionJSONString = process.argv.slice(2);

var subscription = JSON.parse(subscriptionJSONString);

webPush.setGCMAPIKey('YOUR SERVER KEY');

webPush.sendNotification(subscription.endpoint, {
  userPublicKey: subscription.keys.p256dh,
  userAuth: subscription.keys.auth,

  // TODO 7 - add VAPID object

  payload: JSON.stringify({
    'title': 'First push message!',
    'body': 'From a server!',
    'primaryKey': '-push-notification'
  })
})
.then(function(r) {
  console.log('Pushed message successfully!', r);
})
.catch(function(e) {
  console.log('Error', e);
});
```

To test this code, Shift+reload the browser so the service worker updates. Click **Show Subscription Object** and copy the whole subscription object. Paste the subscription object into the command below.

Open a new command window **at your app's base directory** (or stop the Python server with Ctrl+C and use that window) and enter the command below. Replace `SUBSCRIPTION_OBJECT` with your subscription object.

```
node node/main.js 'SUBSCRIPTION_OBJECT'
```

A push notification should popup on the screen. You should also see `Pushed message successfully!` display in the command window in case the notification doesn't pop up immediately.

Explanation

We are using the web-push Mozilla library for Node.js to simplify the process of sending a message to the push service. The code we added to **node/main.js** gets the subscription object, parses it, and sets the Server key. It then passes the subscription endpoint to the `sendNotification` method and passes the public keys and payload to the 'options' object in the second argument.

For more information

web-push library documentation

Other Web Push libraries

Solution code

```
$ git reset --hard
$ git checkout 05-8-payload
```

5.9 Identify your service with VAPID Auth (optional)

Let's add a VAPID object to your push message to identify your app for the push service.

First, replace TODO 6 in **node/main.js** with the code to generate the keys:

node/main.js

```
var serviceKeys = webPush.generateVAPIDKeys();
```

Next, replace TODO 7 in the options parameter on the sendNotification method with a VAPID object that includes the parameters required for the request signing:
Remember to add your email to the subject property in the VAPID object.

node/main.js

```
vapid: {
  subject: 'YOUR EMAIL',
  publicKey: serviceKeys.publicKey,
  privateKey: serviceKeys.privateKey
},
```

You can test what we've done in the browser to make sure everything is working, but the changes we have made in this section won't be visible.

Explanation

Both Chrome and Firefox support the The Voluntary Application Server Identification for Web Push (VAPID) protocol for the identification of your service.

The web-push library makes using VAPID relatively simple, but the process is actually quite complex behind the scenes. For a full explanation of VAPID, see the link in the "For more information" section below.

Now, in our code in **node/main.js**, we are generating the keys every time we run the program. This process is very resource intensive. In a real-world app, you only want to do this once and store the keys somewhere safe.

For more information

Using VAPID

Solution code

```
$ git reset --hard
$ git checkout 05-9-vapid
```

6. Best practices (optional)

Checkout the code to get set up for this section:

```
$ git reset --hard
$ git checkout 06-0-best-practices-setup
```

6.1 Manage the number of notifications

Replace TODO 1 and TODO 2 in the `displayNotification` functions with the code to show two different notifications. Give the first a tag attribute of 'id1' and the second 'id2'.
Test the code in the browser. Click **Show Notification 1** and **Show Notification 2** multiple times each. You should see notifications with the same tags replacing themselves instead of creating new notifications.

Explanation

Whenever you create a notification with a tag and there is already a notification with the same tag visible to the user, the system automatically replaces it without creating a new notification.

Your can use this to group messages that are contextually relevant into one notification. This is a good practice if your site creates many notifications that would otherwise become overwhelming to the user.

Solution code

```
$ git reset --hard
$ git checkout 06-1-multiple-notifications
```

6.2 When to show notifications

If the user is already using our application there's no need to send them notifications.

Replace TODO 3 in **sw.js** with the following code:

sw.js

```
self.addEventListener('push', function(e) {
  clients.matchAll().then(function(c) {
    if (c.length == 0) {
      // Show notification
      e.waitUntil(
        self.registration.showNotification('Push notification')
      );
    } else {
      // Send a message to the page to update the UI
      console.log('Application is already open!');
    }
  });
});
```

We can test the above code by pushing a message from the server. Replace the Server key in node/main.js with the Server key from your own project on Firebase:

node/main.js

```
webPush.setGCMAPIKey('YOUR_SERVER_KEY');
```

Replace the `gcm_sender_id` in manifest.json with your Sender Id:

manifest.json

```
"gcm_sender_id": "YOUR_SENDER_ID"
```

Reload the app in the browser to install the new service worker. Click **Show Subscription Object** and copy the object that appears on the page.

Past the object into the command below and run it in the command window at the base directory of your app:

```
node node/main.js 'SUBSCRIPTION_OBJECT'
```

Send the message once with the app open, and once without. With the app open, the notification should not appear, and instead a message should display in the console. With the application closed, a notification should display normally.

Explanation

The `clients` global in the service worker lists all of the active push clients on this machine. If there are no clients active, the user must be in another app. You should send a notification.

If there *are* active clients it means that the user has your site open in a number of windows. The best practice is usually to relay the message to each of those windows.

Solution code

```
$ git reset --hard
$ git checkout 06-2-when-to-show-notifications
```

6.3 Hide notifications on page focus

If there are several open notifications originating from our app, close them all when the user clicks on one.

In **sw.js**, in the notificationclick event handler, replace the TODO 4 with the following code:

sw.js

```
self.registration.getNotifications().then(function(notifications) {
  notifications.forEach(function(notification) {
    notification.close();
  });
});
```

Optional: If you don't want to clear out all of the notifications, you can filter based on the tag by passing the tag into the `getNotifications` function. Try declaring an 'options' variable inside the notificationclick handler and give it the value `{tag: 'id2'};`. Pass the 'options' variable into the getNotifications method. Reload the service worker in the browser and test it out.

Test the code in the browser. Reload the page and install the new service worker. Remove the `tag` from one of the notifications click the corresponding button on the page a few times to display multiple notifications. If you click one they should all disappear.

You could also filter out the notifications directly inside the promise returned from `getNotifications`. For example there might be some custom data attached to the notification that you would use as your filter-criteria.

Explanation

In most cases, you send the user to the same page that has easy access to the other data that is held in the notifications. Therefore it is wise to clear out all of the notifications that we have created by iterating over the notifications returned from `getNotifications()` method on our service worker registration and then closing each notification.

Solution code

```
$ git reset --hard
$ git checkout 06-3-hide-notifications
```

6.4 Notifications and tabs

We can re-use existing pages rather than opening a new tab when the notification is clicked.

To complete TODO 5, update the `notificationclick` handler to look like this:

sw.js

```
self.addEventListener('notificationclick', function(e) {
  var notification = e.notification;
  var primaryKey = notification.data.primaryKey;
  var action = e.action;

  if (action === 'close') {
    notification.close();
  } else {
    clients.matchAll().then(function(clis) {
      var client = clis.find(function(c) {
        c.visibilityState === 'visible';
      });
      if (client !== undefined) {
        client.navigate('samples/page' + primaryKey + '.html');
        client.focus();
      } else {
        // there are no visible windows. Open one.
        clients.openWindow('samples/page' + primaryKey + '.html');
        notification.close();
      }
    });
  }

  var options = {tag: 'id2'};

self.registration.getNotifications(options).then(function(notifications) {
    notifications.forEach(function(notification) {
      notification.close();
    });
  });
});
```

Reload the service worker in the browser. Get the subscription object and send a notification from the Node.js server. Try clicking on a notification once with the app open in the browser and once without.

When the user clicks on the notification, we get a list of all the open clients and then it is up to us to decide which one you would like to change the page on.

Solution code

```
$ git reset --hard
$ git checkout 06-4-reuse-tabs
```

7. Congratulations

In this course we have learned how to create notifications and configure them so that they look great on the user's device. We have also learned how to build notifications that the user can interact with either via a single tap, or a click on one of a number of different actions.

We have explored how to send messages to the user's device irrespective of whether they have the browser open through the Open Web Push protocol and how to implement this across all the browsers that support this API.

Push notifications are an incredibly powerful mechanism to keep in contact with your users and combined with the low friction of accessing a web site it has never been easier to build up meaningful relationships with your customer. By following this guidance you will be able to build a great experience that your users love and keep coming back to.

8. More resources

Your first push notifications

https://developers.google.com/web/fundamentals/getting-started/push-notifications/?hl=en

When to use push notifications

https://developers.google.com/web/fundamentals/engage-and-retain/push-notifications/?hl=en

Simple Push Demo

https://gauntface.github.io/simple-push-demo/

Notification generator

https://tests.peter.sh/notification-generator/#actions=8

Web Push Libraries

https://github.com/web-push-libs

Encryption

https://developers.google.com/web/updates/2016/03/web-push-encryption?hl=en

Firebase Cloud Messaging

https://firebase.google.com/docs/cloud-messaging/

https://firebase.google.com/docs/cloud-messaging/chrome/client

Lab: Building the checkout workflow

Lab: Integrating Analytics

This tutorial shows you how to integrate Google Analytics into your apps.

Contents:

Overview

What you will learn

- How to create a Google Analytics account
- How to integrate Google Analytics into a web app
- How to add and track custom events (including push notifications)
- How to use analytics even when offline

What you should know

- Basic JavaScript and HTML
- The concept of an Immediately Invoked Function Expression (IIFE)

- How to clone GitHub repos and checkout branches from the command line
- How to enable the developer console

What you will need

- Computer with terminal/shell access
- Connection to the internet
- Chrome browser
- A text editor
- A GitHub account
- A Google account

1. Setting up

Download and unzip the repository, **google-analytics-lab.zip**. Change to the current working directory with the following commands:

```
$ cd google-analytics-lab
```

This repository contains multiple files:

- *README.md* is documentation for GitHub, and can be ignored
- Sample resources that we use in testing
  - *Images*
  - *page1.html, page2.html, other.html*
- *index.html* is the main HTML page for our sample site/application
- *main.js* is the main JavaScript file for our site/application
- *service-worker.js* is the JavaScript file that is used to create our service worker
- *manifest.json* is used for push services
- *node_modules* contains a library for offline analytics

2. Start a local web server and load the page

Checkout the starting code branch:

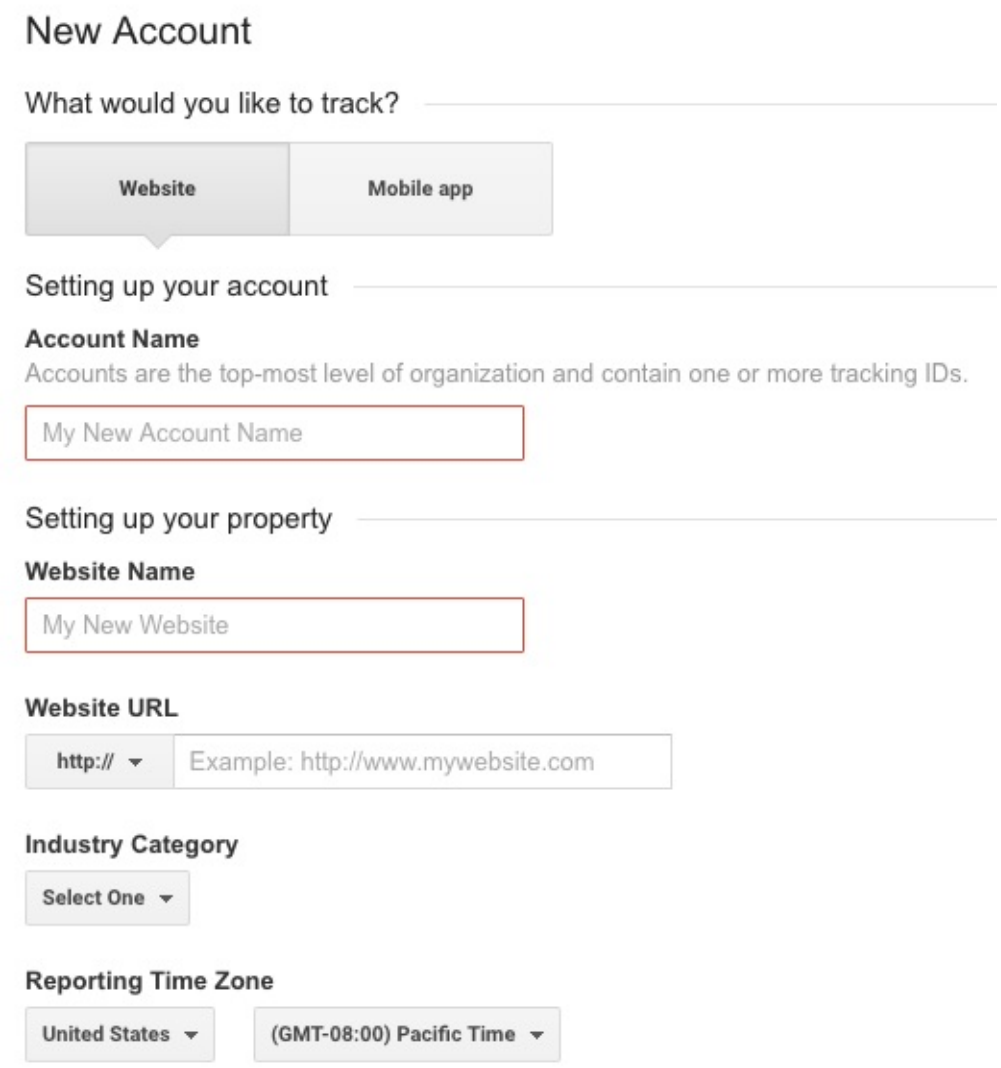
```
$ git checkout 02_setting-up
```

Start a local web server in the project directory. Then, open your browser and navigate to the appropriate local host port (for example, *http://localhost:8000/*).

3. Create a Google Analytics account

In a separate tab or window, navigate to the Google Analytics home page. Sign into your Google/Gmail account, and then select “Sign up” to begin creating your Google Analytics account.

Note: If you already have Google Analytics as part of your Google/Gmail account, select the Admin tab. Under “account”, select your current Google Analytics account and choose “create new account”. A single Google/Gmail account can have multiple Google Analytics accounts.
Your screen should look like this



What would you like to track?

Choose website. Websites and mobile apps implement Google Analytics differently. This lab covers web sites. For mobile apps, see analytics for mobile applications.

Setting up your account

Enter an account name, for example “PWA Training”.

Setting up your property

The property must be associated with a site. We will use a GitHub Pages site.

Note: This is effectively just a placeholder, you do not need to worry if you are unfamiliar with GitHub Pages.
Set the website name to whatever you want, for example “GA Code Lab Site”.

Set the website URL to ***username.github.io/google-analytics-lab/***, where ***username*** is your GitHub username. Set the protocol to *https://*.

Select any industry or category.

Select your time zone.

Unselect any data sharing settings.

Then choose “Get Tracking ID” and agree to the terms and conditions.

Explanation

Your account is the top most level of organization. For example, an account might represent a company. An account has properties that represent individual collections of data. One property in an account might represent the company’s web site, while another property might represent the company’s iOS app. These properties have tracking ID’s (also called property ID’s) that identify them to Google Analytics. You will need to get the tracking ID to use for your app.

For more information

Analytics for mobile applications

GitHub and GitHub Pages

Properties

Google/Gmail accounts

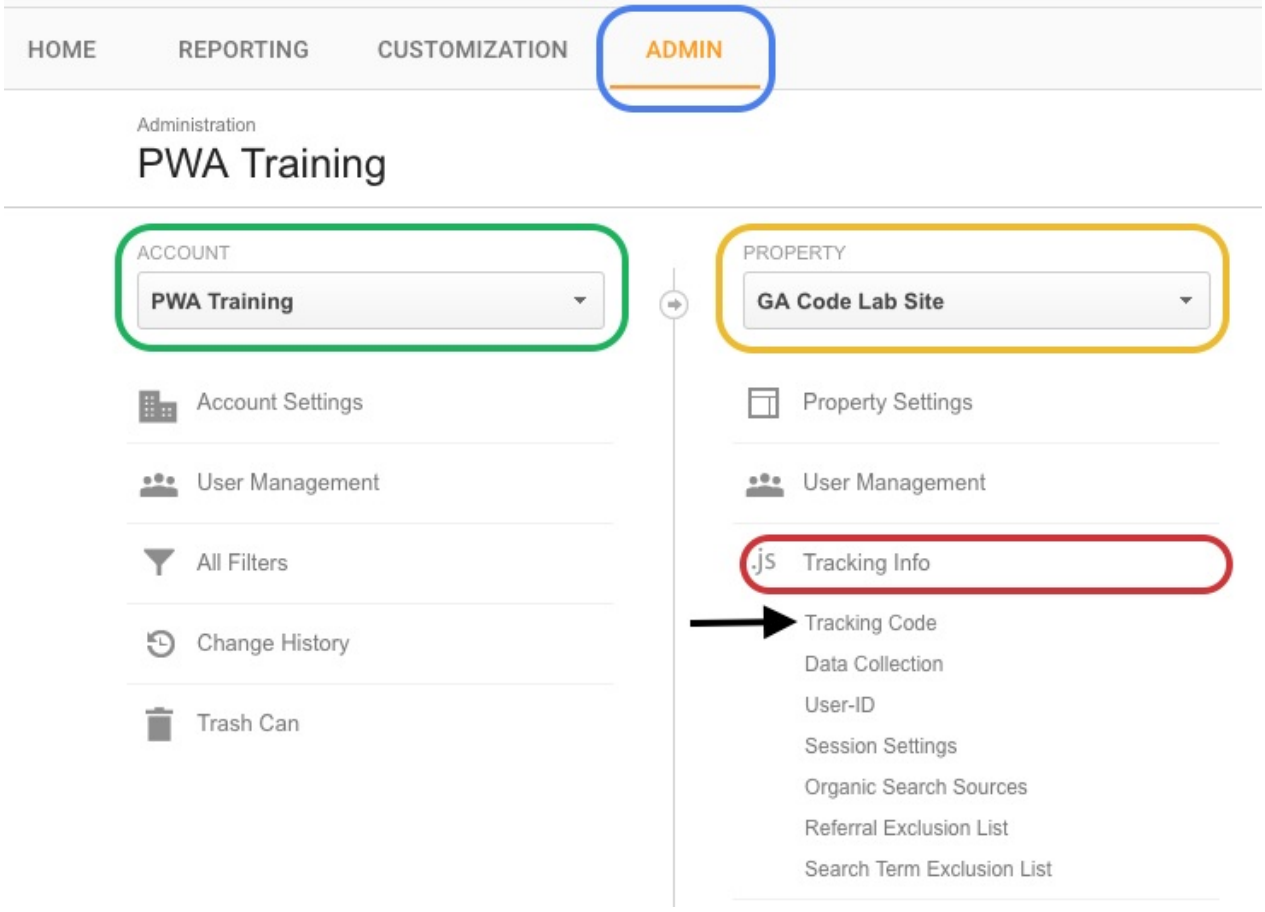
Google Analytics

4. Get your tracking ID and snippet

You should now see your property's tracking ID and tracking code snippet.

If you lost your place:

1. Select the Admin tab
2. Under "account", select your account (for example "PWA Training") from the drop down list.
3. Then under "properties", select your property (for example "GA Code Lab Site") from the down list.
4. Now choose "tracking info", followed by "tracking code".

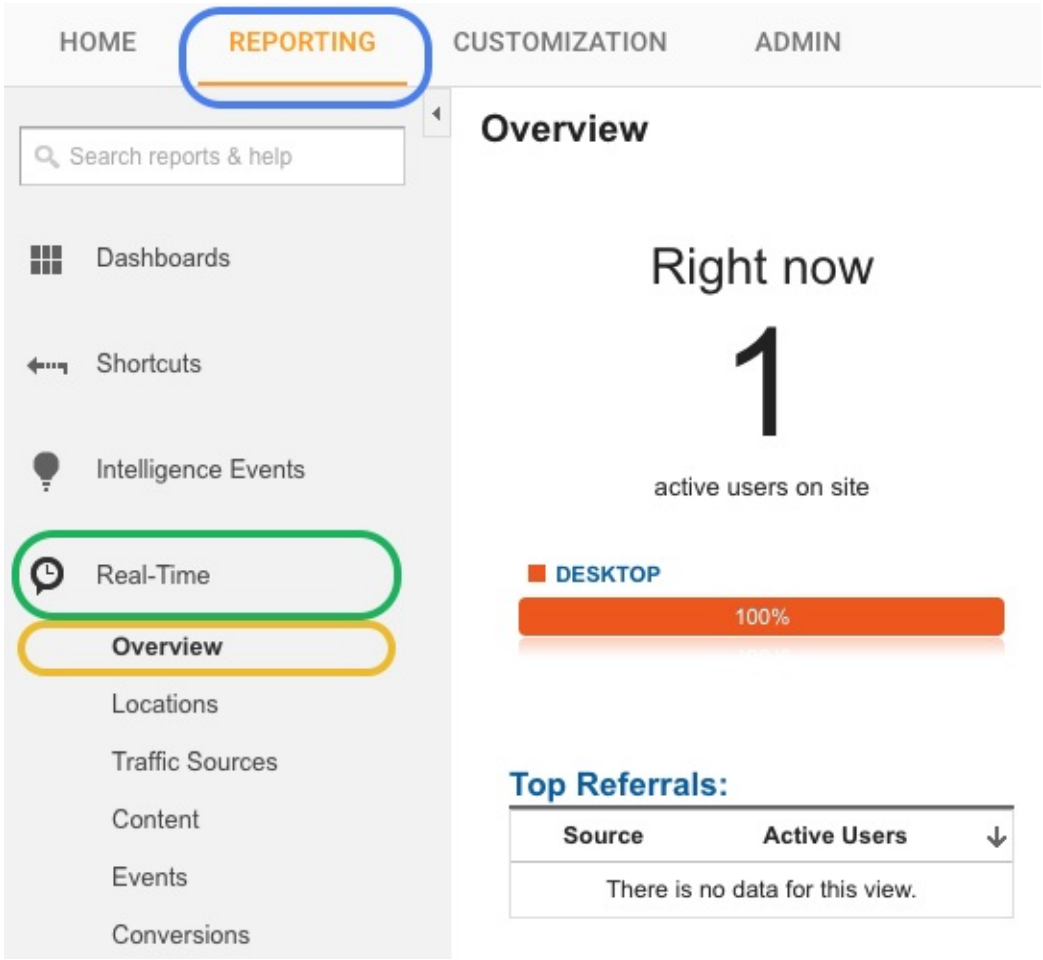


Your tracking ID looks like **UA-XXXXXXXX-Y** and your tracking code snippet looks like:

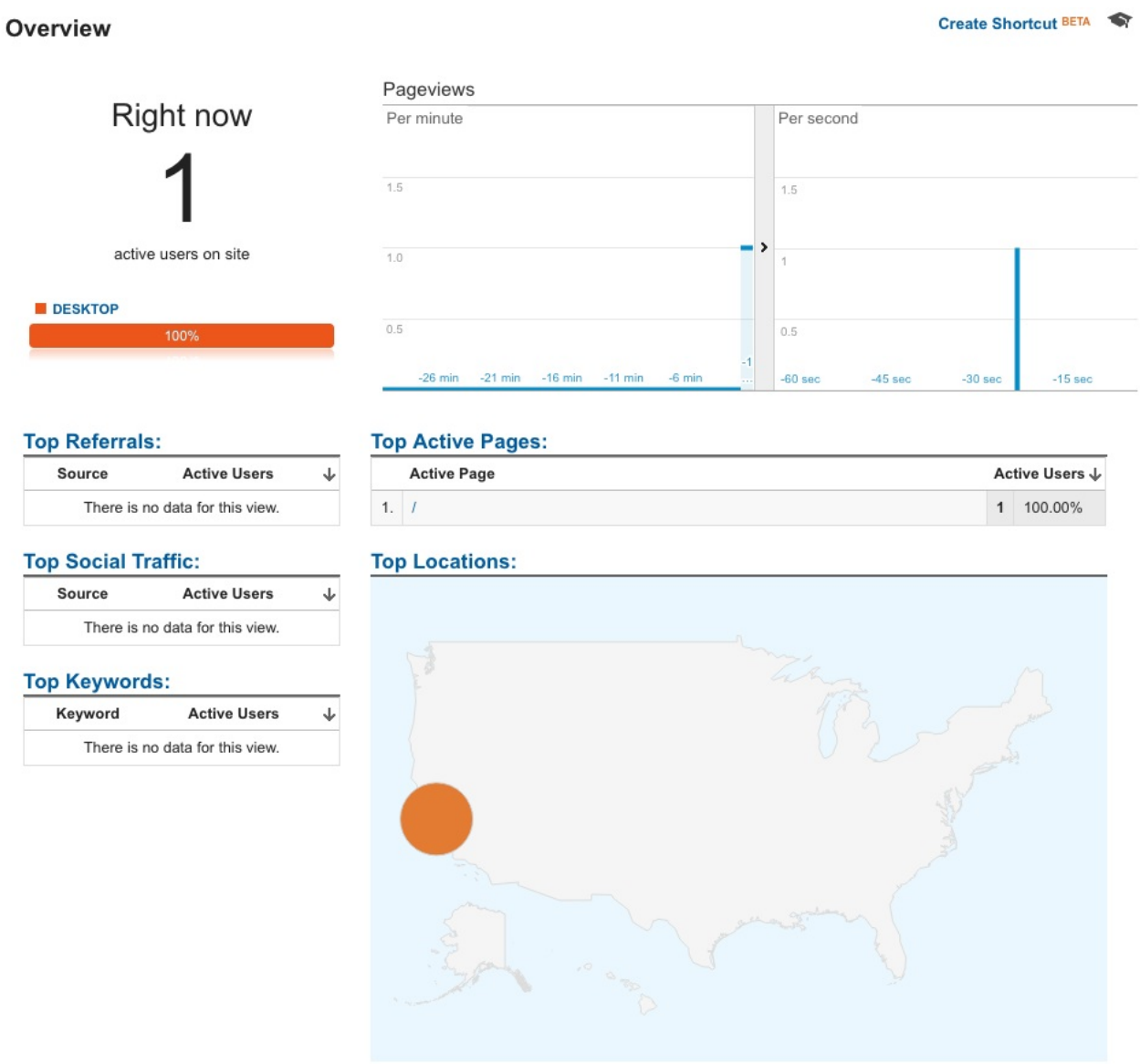
```
<script>
  (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){(i[r].q=
i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),m=s.getElementsB
yTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)})(window,document,'scr
ipt','https://www.google-analytics.com/analytics.js','ga');

  ga('create', 'UA-XXXXXXXX-Y', 'auto');
  ga('send', 'pageview');

</script>
```

Copy this script and paste it in the TODO 4 in *index.html* and *other.html*. Save the scripts and refresh the page at localhost. Return to the Google Analytics site. Examine the real time data:

1. Select the Reporting tab
2. Select Real-Time
3. Select Overview



You should see yourself being tracked. The screen should look similar to this:



Note: If you don't see this, refresh the page at localhost a couple times.
Observe that the "Active page" indicates which page is being viewed. On localhost, navigate to the other page by clicking the "Other page" link. Then to return to the Google Analytics site and check "Active Page" again. It should now show "/pages/other.html" (this might take a few seconds).

Note: Google Analytics does not always receive every data point, so it is possible that the dashboard won't report this hit in real time.

Explanation

When the page loads, the tracking snippet script is executed. The IIFE in the script does two things

1. Creates another `<script>` tag that starts asynchronously downloading *analytics.js*, the library that does all of the analytics work.

2. Initializes a global `ga` function, called the command queue. This function allows "commands" to be scheduled and run once the *analytics.js* library has loaded.

Note: The snippet code is complicated. The following snippet is a clearer alternative:

```
<script>
window.ga=window.ga||function(){(ga.q=ga.q||[]).push(arguments)};ga.l=+new Date;
ga('create', 'UA-XXXXX-Y', 'auto');
ga('send', 'pageview');
</script>
<script async src='https://www.google-analytics.com/analytics.js'></script>
```

The difference is that this new code relies on the `async` tag, which is not supported in some older browsers. You can learn more about the tracking snippet.
The next lines add two commands to the queue. The first creates a new tracker object. Tracker objects track and store data. When the new tracker is created, the analytics library gets the user's IP address, user agent, and other page information, and stores it in the tracker. From this info Google Analytics can extract:

- User's geographic location
- User's browser and operating system (OS)
- Screen size
- If Flash or Java is installed
- The referring site

The second command sends a "hit". This sends the tracker's data to Google Analytics. Sending a hit is also used to note a user interaction with your app. The user interaction is specified by the hit type, in this case a "pageview". Since the tracker was created with your tracking ID, this data is sent to your account and property.

The main interface for using *analytics.js* is the `ga` command queue. The command queue stores commands (in order) until *analytics.js* has loaded. Once *analytics.js* has loaded, the commands are executed sequentially. All commands called after *analytics.js* has loaded execute immediately. This functionality ensures that analytics can begin independent of the loading time of *analytics.js*.

Commands are added by calling `ga()`. The first argument passed is the command, which is a method of the *analytics.js* library. The remaining arguments are parameters for that method.

In this code, `'create'` is the command. `'UA-XXXXX-Y'` and `'auto'` are parameters for the `'create'` method. The `ga()` signature is flexible. The create command could also be written as:

```
ga('create', {
  trackingId: 'UA-XXXXX-Y',
  cookieDomain: 'auto'
});
```

Unknown commands are ignored, and will not throw an error.

The code so far provides the basic functionality of Google Analytics. A tracker is created and a pageview hit is sent every time the page is visited. In addition to the data gathered by tracker creation, the pageview event allows Google Analytics to infer:

- The total time the user spends on the site (usually requires a hitCallback)
- The time spent on each page and the order the pages are visited
- Which internal links are clicked (based on the URL of the next pageview)

Real-time mode in the Google Analytics dashboard shows the hit received from this script execution, along with the page (under “Active Page”) that it was executed on (for example “/” or “/index.html”).

For more information

The tracking snippet

Tracker objects

Creating trackers

The create command

The send command

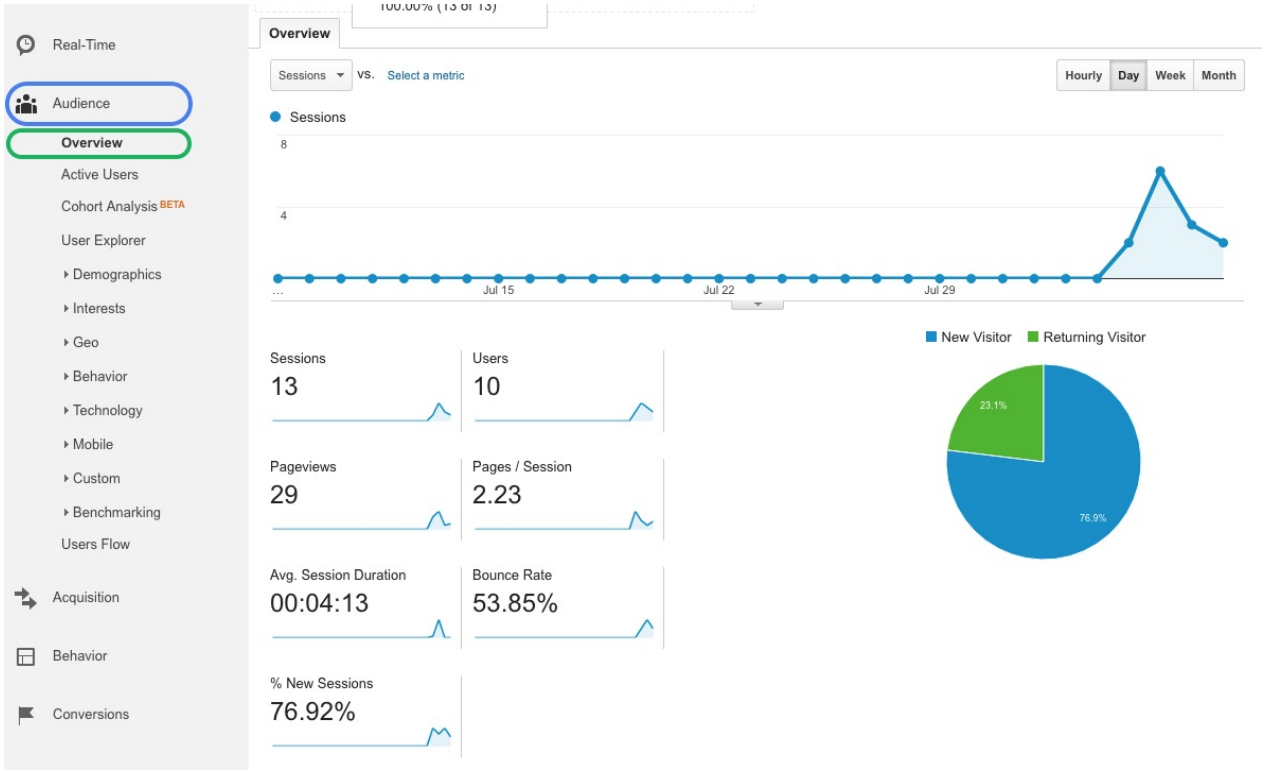
Hits

The data sent in a hit

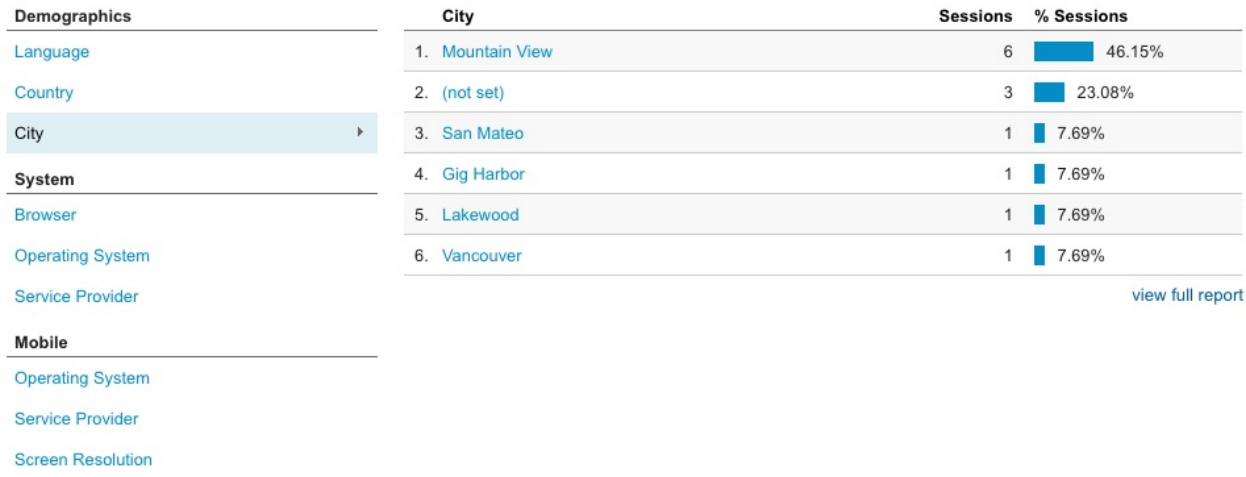
5. Viewing user data

We are using the real-time viewing mode because we have just created the app. Normally, records of past data would also be available. You can view this from the reporting tab by selecting Audience and then Overview.

Here you can see general information such as pageview records, bounce rate, ratio of new and returning visitor, and other statistics.



As well as specific information like visitors' language, country, city, browser, operating system, service provider, screen resolution, and device.



For more information

Learn about Google Analytics for business

6. Debugging

Checking the dashboard is not an efficient method of testing. Google Analytics offers the *analytics.js* library with a debug mode.

Replace `analytics.js` in the tracking snippet (in *index.html* and *other.html*) with `analytics_debug.js`.

Save the scripts and refresh the page. Open the console. You should see logs detailing the “create” and “send” commands.

Note: Don’t use *analytics_debug.js* in production. It is much larger than *analytics.js*.
Note: There is also a Chrome debugger extension that can be used alternatively.
Navigate back to index.html using the “back” link. Note how the location field changes on the data sent in the send command.

For more information

Chrome debugger extension

Debugging Google Analytics

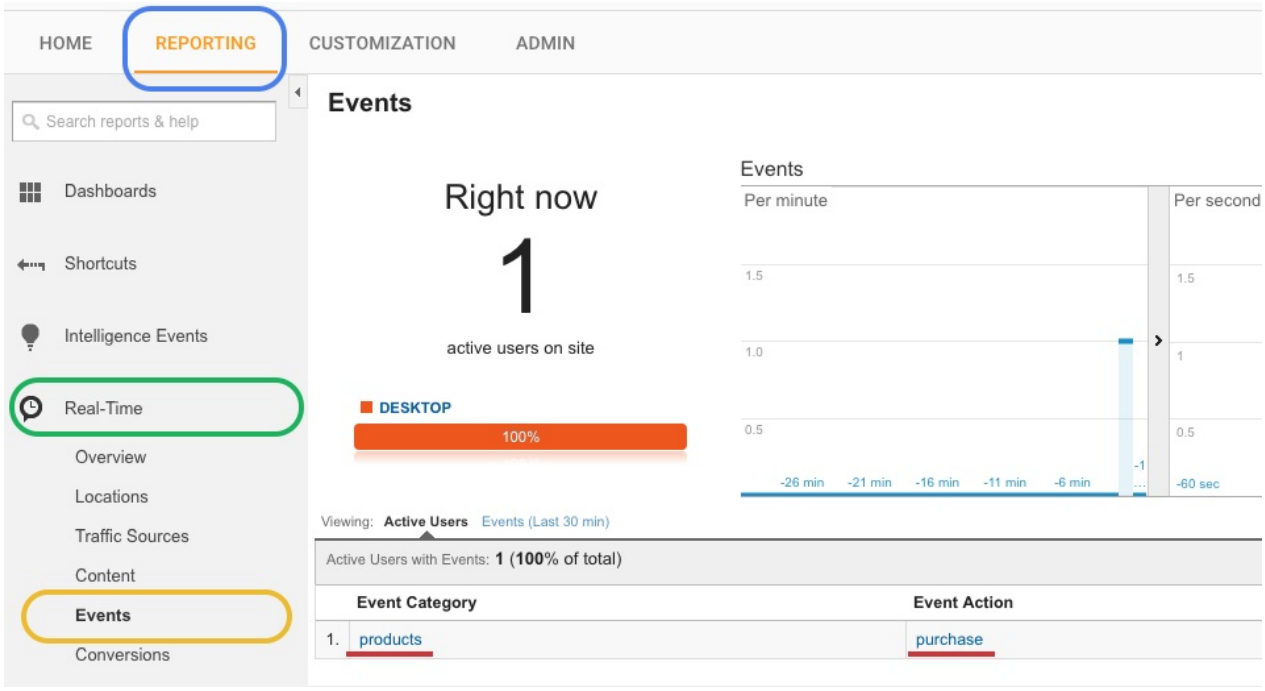
7. Custom events

Google Analytics supports custom events that allow fine grain analysis of user behavior.

In *main.js*, replace TODO 7 with

```
ga('send', {
  hitType: 'event',
  eventCategory: 'products',
  eventAction: 'purchase',
  eventLabel: 'Summer products launch'
});
```

Save the script and refresh the page. Click the “BUY NOW!!!” button. Check the console, do you see the custom event? Now return to the real-time reporting section of the Google Analytics dashboard (from the Reporting tab, select Real-Time). Instead of selecting Overview, select Events. Do you see the custom event?



Explanation

When using the send command in the `ga` command queue, the hit type can be set to ‘event’, and values associated with an event can be added as parameters. These values represent the eventCategory, eventAction, and eventLabel. All of these are arbitrary, and used to organize events. Sending these custom events allows us to deeply understand user interactions with our site.

Note: Many of the `ga` commands are flexible and can use multiple signatures. For example

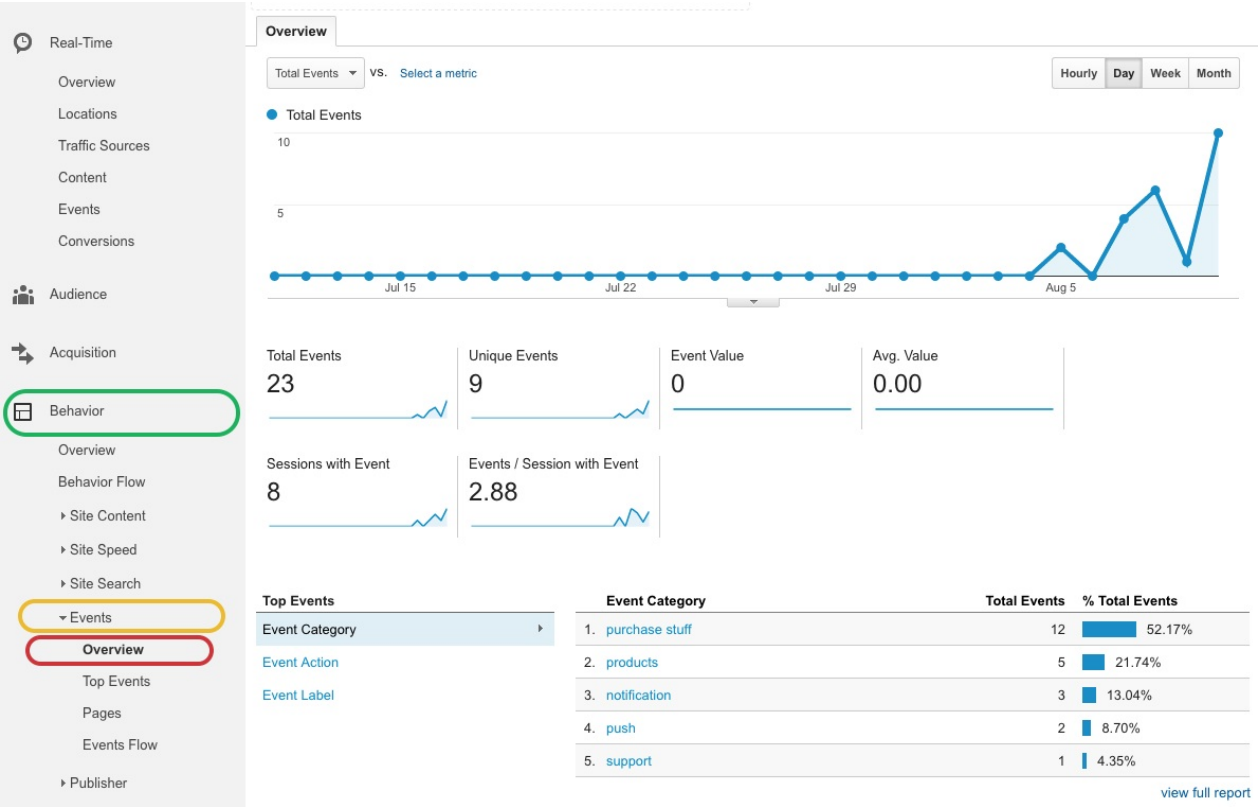
```
ga('send', {
  hitType: 'event',
  eventCategory: 'Videos',
  eventAction: 'play',
  eventLabel: 'Fall Campaign'
});
```

Can also be written:

```
ga('send', 'event', 'Videos', 'play', 'Fall Campaign');
```

You can see all method signatures in the command queue reference.
Optional: Update the custom event that you just added to use the alternative signature.

You can view past events in the Google Analytics dashboard from the Reporting tab by selecting Behavior, followed by Events and then Overview.



For more information

Event tracking

About events

Command queue reference

8. Notifications

Google Analytics can be used to understand user experience with notifications.

8.1 Add notifications

We will begin by adding notifications (non-push) to the app.

In *main.js* replace TODO 8a with:

```
var registration; // Store service worker registration
navigator.serviceWorker.register('service-worker.js')
.then(function(reg) {
  console.log('Service worker registered', reg);
  registration = reg;
})
.catch(function(error) {
  console.log('Service Worker registration failed:', error);
});
```

Replace the TODO 8b with:

```
// Browser may prompt user with permission request
Notification.requestPermission(function(status) {
  console.log('Notification permission status:', status);
});
```

Replace TODO 8c with:

```
var notifyButton = document.getElementById('notify');
notifyButton.addEventListener('click', function() {
  displayNotification();
});
```

Replace TODO 8d with:

```
function displayNotification() {
  if (Notification.permission == 'granted') {
    var options = {
      body: 'You have a new notification!',
      icon: 'images/notification-flat.png',
      vibrate: [100, 50, 100],
      data: {
        dateOfArrival: Date.now(),
        primaryKey: 1
      },
      actions: [
        {action: 'open', title: 'Open the site!',
          icon: 'images/checkmark.png'},
        {action: 'close', title: 'Go away!',
          icon: 'images/xmark.png'},
      ]
    };
    registration.showNotification('Hello world!', options);
    // TODO Step 8e: send notification display event
  } else {
    console.warn('Notifications are blocked');
    // TODO Step 8f: send notification blocked event
  }
}
```

Save the script and refresh the page. Accept the prompt to display notifications (if you are not prompted, manually set notifications to be allowed by clicking the icon left of the site URL). Now click the “Manual notification” button. You should be prompted with a notification. Close it by clicking the x in the upper right hand corner.

Explanation

We started by registering the service worker at *service-worker.js*. This file is currently an empty IIFE, but service workers are required for notifications. Next we add a user prompt for notification permission. We then attach a function to the “Manual notification” button. If notifications are permitted, the function creates and displays a notification (the notification actions don’t work yet). If notifications are blocked, a console warning is shown.

8.2 Adding analytics to notifications

Replace TODO 8e with:

```
ga('send', 'event', 'notification', 'displayed-standard');
```

Replace TODO 8f with:

```
ga('send', 'event', 'notification', 'blocked');
```

Save the script and refresh the page. Click the “Manual notification” button. You should see a custom event shown in the console. Confirm that the event was sent by looking at the real-time events back on the Google Analytics page.

Manually block notifications on the localhost page by clicking on the icon just left of the URL and revoking permission for notifications. Refresh the page and click the “Manual notification” button. The console should warn that notifications are blocked, and you should see that a different custom event was sent. Confirm by examining the real-time events on the Google Analytics site.

Explanation

We have added Google Analytics send commands inside our notification code. This can allow us to track how often users are receiving notifications, and how many of those users are blocking notifications for our site.

Optional: Replace TODOs 8x and 8y with send commands indicating that the browser is missing the appropriate support. Test in an unsupported browser, and examine the console logs and Google Analytics site. This could be used to tell us how many of our users aren’t receiving our notifications because of lack of support.

8.3 Interacting with notifications

Our notifications do not yet support actions.

In *service-worker.js* replace TODO 8g with:

```
self.addEventListener('notificationclose', function(event) {
  var notification = event.notification;
  var primaryKey = notification.data.primaryKey;
  console.log('Closed notification:', primaryKey);
  // TODO Step 8k-1: send notification close event
});
```

Replace TODO 8h with:

```
self.addEventListener('notificationclick', function(event) {
  var notification = event.notification;
  var primaryKey = notification.data.primaryKey;
  var action = event.action;
  if (action === 'close') {
    notification.close();
    console.log('Closed notification:', primaryKey);
    // TODO Step 8k-2: send notification close event
  } else if (action === 'open') {
    clients.openWindow('pages/page' + primaryKey + '.html');
    notification.close();
    console.log('Notification opened');
    // TODO Step 8l: Send notification opened event
  } else {
    clients.openWindow('pages/page' + primaryKey + '.html');
    notification.close();
    console.log('Notification clicked');
    // TODO Step 8m: Send notification clicked event
  }
});
```

Save the script. Unblock notifications if you have not already done so (by clicking the icon just left of the site URL) and refresh the page (this also installs the new service worker). Hold shift & refresh the page to perform a hard reload, allowing the updated service worker to take over. Click the “Manual notification” button. The notification actions should now function.

Test the notification actions by clicking both actions, as well as clicking on the notification itself (not one of the actions).

Explanation

We have notification functionality. Now we want to send Google Analytics events when these notifications are closed, opened, and clicked. However, the service worker script runs on its own thread and doesn’t have access to the `ga` command queue object (which is on the main thread). We need a way to communicate between the service worker and the main thread.

8.4 Communicating with the service worker

In *main.js* replace TODO 8i with:

```
function openCommunication() {
  var msgChannel = new MessageChannel();
  msgChannel.port1.onmessage = function(msgEvent) {
    sendAnalytics(msgEvent.data);
  };
  navigator.serviceWorker.controller.postMessage(
    'Establishing contact with service worker', [msgChannel.port2]
  );
}
openCommunication();

function sendAnalytics(message) {
  var eventCategory = message.eventCategory;
  var eventAction = message.eventAction;
  ga('send', 'event', eventCategory, eventAction);
}
```

In *service-worker.js* replace TODO 8j with:

```
var mainClientPort;
self.addEventListener('message', function(event) {
  console.log('Service worker received message: ', event.data);
  mainClientPort = event.ports[0];
  console.log('Communication ports established');
});
```

Save the scripts and refresh the page (to install the updated service-worker). Close all localhost pages, and reopen the main page. This will allow the updated service worker to take over. Check the console. You should see logs indicating that communication has been established with the service worker.

Note: A hard reload is not adequate now that *main.js* calls `serviceWorker.controller`, which returns null on hard reloads.

Explanation

The code we add to *main.js* creates a MessageChannel, which allows us to send data across threads on its two MessagePorts. On the first MessagePort (port1), we attach a message listener. When the message listener receives a message (which will come from the service worker), it calls a `sendAnalytics` function with the message data. The `sendAnalytics` function parses the data and executes a Google Analytics send command just like we have done previously. A message is sent to the service worker with the postMessage function, containing the MessageChannel's second MessagePort (port2).

The code we add to *service-worker.js* adds a generic listener (not on any specific MessageChannel or MessagePort) to the service worker to receive this message. When the message is received from *main.js*, the second message port (port2) is saved. This will allow the service worker to send data back to *main.js*.

Now we have the ability for the service worker to send messages to *main.js* (using port2). And *main.js* will hear those messages and send custom events to Google Analytics with the message data.

For more information

MessageChannel

MessagePorts

postMessage

Web workers

Communicating with service worker example

8.5 Analytics for user interactions

Now we can add analytics to user interactions with the notifications.

In *service-worker.js* replace TODO 8k-1 and 8k-2 with:

```
mainClientPort.postMessage({
  eventCategory: 'notification',
  eventAction: 'closed'
});
```

Replace TODO 8l with:

```
mainClientPort.postMessage({
  eventCategory: 'notification',
  eventAction: 'opened'
});
```

Replace TODO 8m with:

```
mainClientPort.postMessage({
  eventCategory: 'notification',
  eventAction: 'clicked'
});
```

Save the script and refresh the page (to install the updated service-worker). Close all localhost pages and reopen the main page. This will allow the updated service worker to take over. Now click the “Manual notification” button and test the functionality of the notifications. Try selecting both actions and clicking on it generally (but not on an action). Notice that a custom event is fired in each case. Confirm this in both the console and the Google Analytics page.

Explanation

Rather than sending data directly to Google Analytics, we are using the `postMessage` function to send data to *main.js*, which then sends the data to Google Analytics.

9. Push notifications

We will now incorporate push into our notifications.

9.1 Incorporating push

Uncomment the *manifest.json* link in TODO 9a in *index.html*. In the *manifest.json* file, add your Firebase Cloud Messaging data just like you did for the push notification lab. It should look something like:

```
{
  "name": "Push Notifications codelab",
  "gcm_sender_id": "XXXXXXXXXXXX"
}
```

In *main.js* replace TODO 9b with:

```
var subscription;
var isSubscribed = false;
var subscribeButton = document.getElementById('subscribe');

subscribeButton.addEventListener('click', function() {
  if (isSubscribed) {
    unsubscribe();
  } else {
    subscribe();
  }
});

function subscribe() {
  registration.pushManager.subscribe({userVisibleOnly: true})
  .then(function(pushSubscription) {
    subscription = pushSubscription;
    console.log('Subscribed!', subscription);
    // TODO Step 9d: send push subscribed event
    subscribeButton.textContent = 'Unsubscribe';
    isSubscribed = true;
  })
  .catch(function(error) {
    if (Notification.permission === 'denied') {
      console.warn('Subscribe failed, notifications are blocked');
      // TODO Step 9e: send push blocked event
    } else {
      console.warn('Error subscribing', error);
      // TODO Step 9f: send push subscribe-error event
    }
  });
}

function unsubscribe() {
  subscription.unsubscribe()
  .then(function() {
    console.log('Unsubscribed!');
    // TODO Step 9g: send push unsubscibe event
    subscribeButton.textContent = 'Subscribe';
    isSubscribed = false;
  })
  .catch(function(error) {
    console.warn('Error unsubscribing', error);
    // TODO Step 9h: send push unsubscribe-error event
    subscribeButton.textContent = 'Subscribe';
  });
}
```

In *service-worker.js* replace TODO 9c with:

```
self.addEventListener('push', function(event) {
  console.log('Push recieved');
  // TODO Step 9i: Send push received event
  if (Notification.permission === 'denied') {
    console.warn('Push notification failed, notifications are blocked');
    // TODO Step 9j: Send push blocked event
    return;
  }
  var options = {
    body: 'This notification was generated from a push!',
    icon: 'images/notification-flat.png',
    vibrate: [100, 50, 100],
    data: {
      dateOfArrival: Date.now(),
      primaryKey: '2'
    },
    actions: [
      {action: 'open', title: 'Open the site!',
        icon: 'images/checkmark.png'},
      {action: 'close', title: 'Go away!',
        icon: 'images/xmark.png'},
    ]
  };
  event.waitUntil(
    Promise.all([
      self.registration.showNotification('Hello world!', options),
      // TODO Step 9k: Send notification displayed-push event
    ])
  );
});
```

Explanation

In *main.js* we are creating subscribe and unsubscribe functions, and attaching them to a UI button. This will allow us to test subscribing and unsubscribing. In *service-worker.js* we add a push listener. This listener will hear push events and generate a notification.

9.2 Adding analytics to push notifications

Now that push is implemented, we can add and test analytics.

In *main.js* replace TODO 9d with:

```
ga('send', 'event', 'push', 'subscribe');
```

Replace TODO 9e with:

```
ga('send', 'event', 'push', 'subscribe-blocked');
```

Replace TODO 9f with:

```
ga('send', 'event', 'push', 'subscribe-error');
```

Replace TODO 9g with:

```
ga('send', 'event', 'push', 'unsubscribe');
```

Replace TODO 9h with:

```
ga('send', 'event', 'push', 'unsubscribe-error');
```

In *service-worker.js* replace TODO 9i with:

```
mainClientPort.postMessage({
  eventCategory: 'push',
  eventAction: 'recieved'
});
```

Replace TODO 9j with:

```
mainClientPort.postMessage({
  eventCategory: 'push',
  eventAction: 'blocked'
});
```

Replace TODO 9k with:

```
mainClientPort.postMessage({
  eventCategory: 'notification',
  eventAction: 'displayed-push'
})
```

Save the scripts and refresh the page. Close all localhost pages and reopen the main page. Perform the following steps and observe the event's logged in the console and on the Google Analytics site:

1. Click the "Subscribe" button to subscribe to push.
2. In dev tools, navigate to the Application tab, and select Service Workers. Click "Push" to

send a simulated push event.
3. Test both of the actions in the notification.
4. Now block notifications (by clicking left of the URL) and refresh the page. Send another simulated push event.
5. Press the “Subscribe button again”.
6. Unblock notifications and refresh the page. Press “Subscribe” again. Now press “Unsubscribe”.
   Note: Simulated push notifications can be sent from the browser even if the subscription object is null.

10. Offline analytics

Analytics data can be stored when users are offline and sent at a later time when they have reconnected.

In *service-worker.js* replace TODO 10 with:

```
importScripts('/node_modules/sw-offline-google-analytics/offline-google-analytics-impo
rt.js');
goog.offlineGoogleAnalytics.initialize();
```

Save the script and refresh the page. Close all localhost pages and reopen the main page. In dev tools, select the Network tab and then select Offline. This will simulate offline behavior.

Trigger some Google Analytics events (for example, click the “BUY NOW!!!” or “Subscribe” buttons). The console logs the events, but they don’t appear in the real-time events section of the Google Analytics site.

Select the Application tab. In Storage, select IndexedDB. Select offline-google-analytics. You should see a list of URLs (you may need to click the refresh icon inside the cache interface). These URL represent the unsent hit requests that occurred while offline.

Return to the Network tab and disable Offline mode. Refresh the page. Now return to the Application and examine the offline-google-analytics cache again. You should see that the URL’s have been cleared (indicating that they have been sent to Google Analytics).

Explanation

Here we import and initialize the *offline-google-analytics-import.js* library. This adds a fetch event handler to the service worker that only listens for requests made to the Google Analytics domain. The handler attempts to send Google Analytics data just like we have done so far, by network requests. If the network request fails, the request is stored in IndexedDB. Each time the service worker starts up again it attempts to resend the stored requests.

For more information

ImportScripts

Offline Google Analytics

Google I/O offline example

IndexedDB

Solution code

To get a copy of the working code, use the following commands:

```
$ git reset --hard
$ git checkout 10_offline
```

Optional: Using hitCallback for custom events

The hitCallback functionality allows you to call a function once a hit has been successfully sent.

Add the following to *main.js*:

```
var link = document.getElementById('external');
link.addEventListener('click', function(event) {
  event.preventDefault();
  ga('send', 'event', 'outbound', 'sponsor1', {
    hitCallback: function() {
      window.location.href = event.target.href;
    }
  });
});
```

Save the script and refresh the page. Click the “External link” link. You should see the custom event logged in the console and on the Google Analytics site.

Explanation

This code adds an event listener to an external link. When the link is clicked, the default action (loading the external page) is prevented. A hit is then sent with the `ga` send command. The send command includes an additional `hitCallback` parameter that specifies a function to be called once the hit is sent successfully. This function then loads the external page (the default action that was prevented).

This situation is common anytime a user action will take the user away from your site. Many browsers stop executing JavaScript once the current page starts unloading. This can prevent `ga` commands from being sent.

Optional: You should always use a timeout function with `hitCallback` when preventing critical actions. If the *analytics.js* library fails to load, the `hitCallback` function will never run and the user will be stuck. Update the previous code to use a timeout that will take the user to the external site even if the `hitCallback` never executes. Test your code to make sure it works.

Note that if the user’s browser supports navigator.sendBeacon then ‘beacon’ can be specified as the transport mechanism. This avoids the need for a hit callback. See the documentation for more info.

For more information

Sending hits

Solution code

To get a copy of the working code, use the following commands: ``` $ git reset --hard $ git checkout optional

Running a local web server

Most of the code labs will require to run from a web server. To meet this requirements we present different ways in which you can run web content locally.

Contents:

What server to use

- If you are running Chrome (on any OS) then install

Web Server for Chrome

- If you're running Chrome OS install

Web Server for Chrome

- If you are running Linux or Macintosh use

Python's Built in Server

Web Server for Chrome

Web Server for Chrome is a Chrome extension that creates a local web server. It will run on all platforms where Chrome runs including Chromebooks.

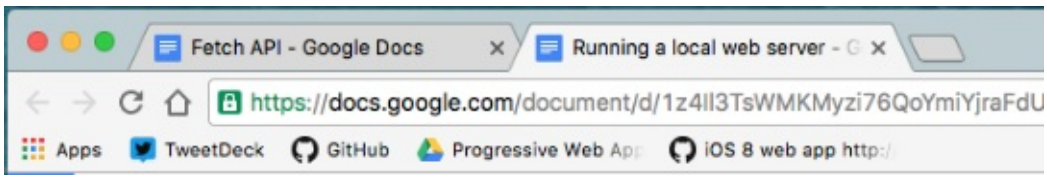
Install Web Server for Chrome from the Chrome Store.

To open the web server extension:

1. Make sure your Bookmark bar is open. Go to the View menu and ensure that Always

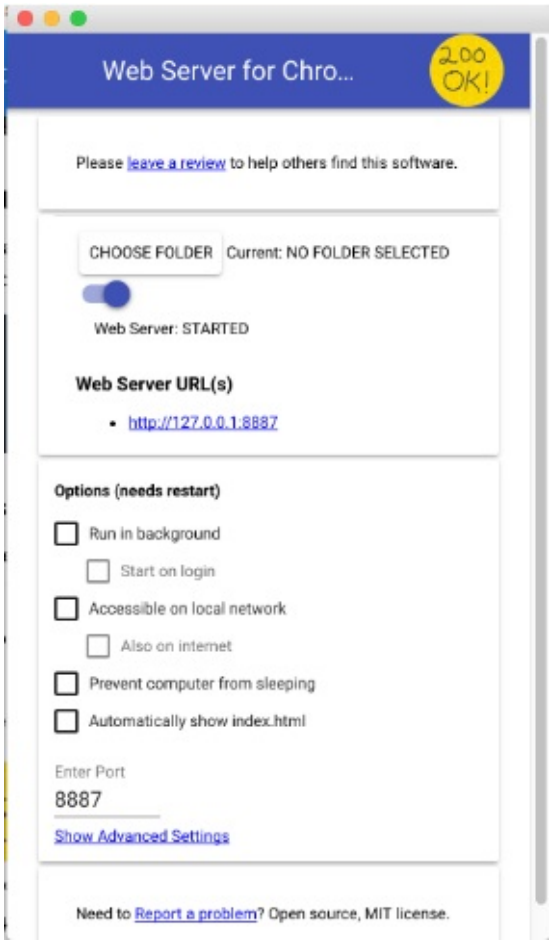
Show Bookmark Bar is checked.

2. Click on the Apps button (far left in the image below)



3. Select the Web Server Icon 

When the web server lunches you will see the following window:



To run your code click ***Choose Folder*** and point to the folder hosting your code and click on the URL under **Web Server URL(s)**. Check 'Automatically show index.html' to test a web app.

Python Web Server

Python has a built in web server that can be executed from your shell or terminal. This will work everywhere Python is installed. versions of Python come preinstalled in Macintosh and most flavors of Linux; Windows users can download an installer from python.org

The exact command will depend on the version of Python installed in your system. To verify what version is installed run the following command:

python -V

Navigate to your app's base directory in a command window.

Python 2

If you have version 2 installed, run the following command:

python -m SimpleHTTPServer 8000

You will be prompted if you want to accept incoming connections. Whatever choice you make is fine.

Python 3

If your version of Python is 3 or higher run:

python -m http.server 8000

You will be prompted if you want to accept incoming connections. Whatever choice you make is fine.

You can now open http://localhost:8000/ in a browser to test your app.

MAMP and XAMPP

MAMP and XAMPP provide web server distribution installers for your operating systems.

MAMP

MAMP is available for both Windows and Macintosh. It installs a local server environment on your computer. To use MAMP to host your content:

- Download the versions for your OS from the MAMP download page

- Copy your project's folder to the root of MAMP's server. In Macintosh the default location is ***/Applications/MAMP/htdocs***. The default Windows location is ***c:\MAMP\htdocs***
- Point your browser to the server URL ***http://localhost:8888/your-folder***

You can find documentation on the documentation site and in the MAMP for Windows documentation site.

XAMPP

XAMPP is a cross platform Apache distribution that installs Apache, MariaDB (fork of MySQL), PHP and Perl on your computer. To use XAMPP to serve your project:

- Download the latest version for your OS from the XAMPP download page
- Copy your content to XAMPP's default document root folder(***C:/xampp/htdocs***)
- Point your browser to your applications (***http://localhost/your-folder***)

Node.js and Gulp

For some code labs we'll make available a Gulp-based build system that includes a browsersync server configured to the correct directory. To start the server run:

gulp server

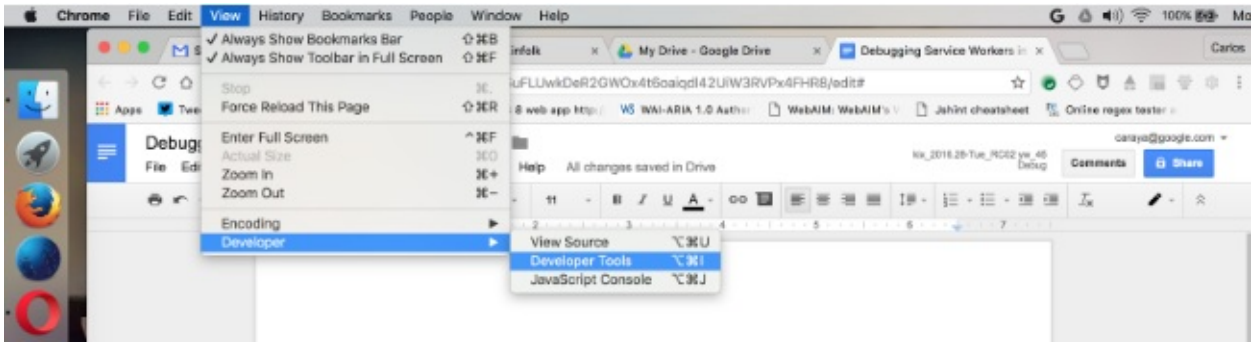
This command will run the server and open your default browser to the index page.

Debugging Service Workers in Browsers

Chrome

Open DevTools with ***F12 or Ctrl + Shift+ I on Windows*** Or ***Cmd + Opt + I on Mac***

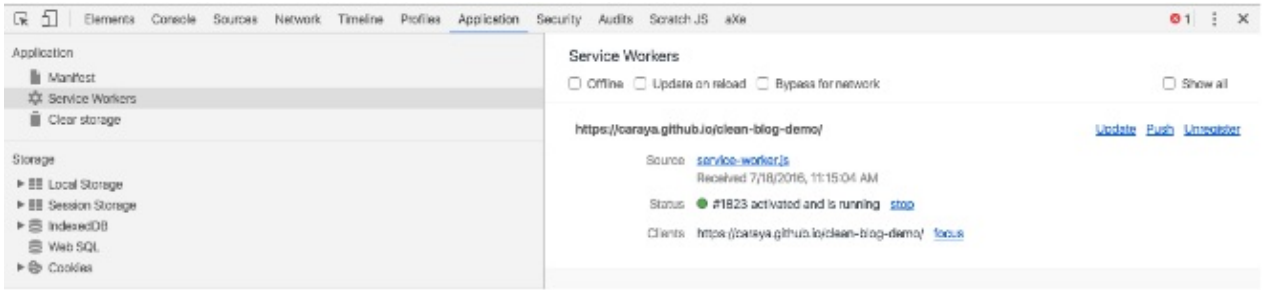
The command is also available under the ***View > Developer > Developer tools*** as shown in the image below:



Once in DevTools you can check the status of your service worker under application. When you click the application tab you will see the current page's service worker in the right side of the screen along with some developer helpers to make working with the service worker easier.

There are three checkboxes:

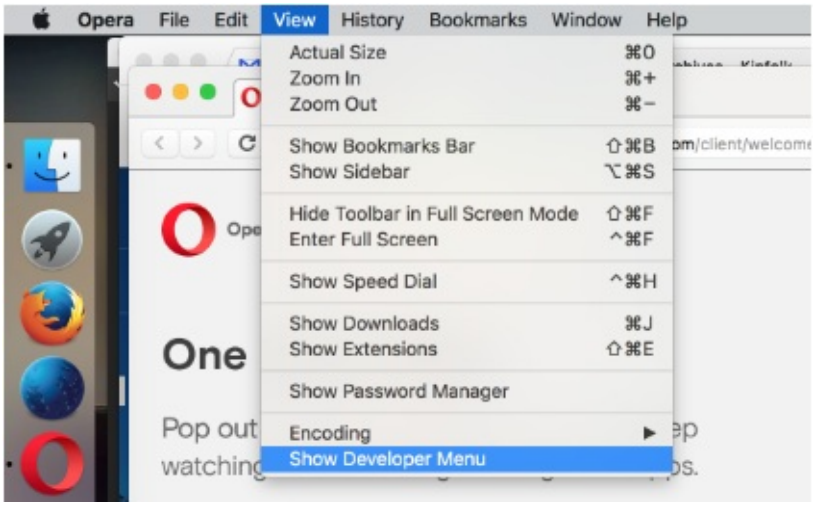
- **Offline**: forces the browser offline, simulating no network connectivity
- **Update on Reload**: reloads the service worker every time the browser reloads, picking up all changes during development
- **Bypass for network**: Ignores the service worker and fetches all resources from the network



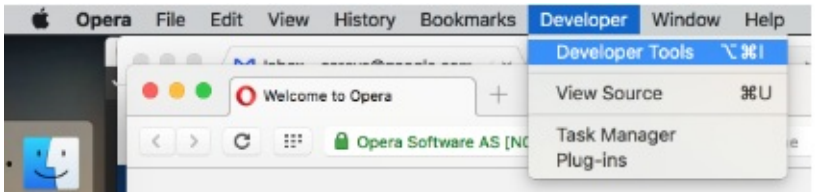
For more information:

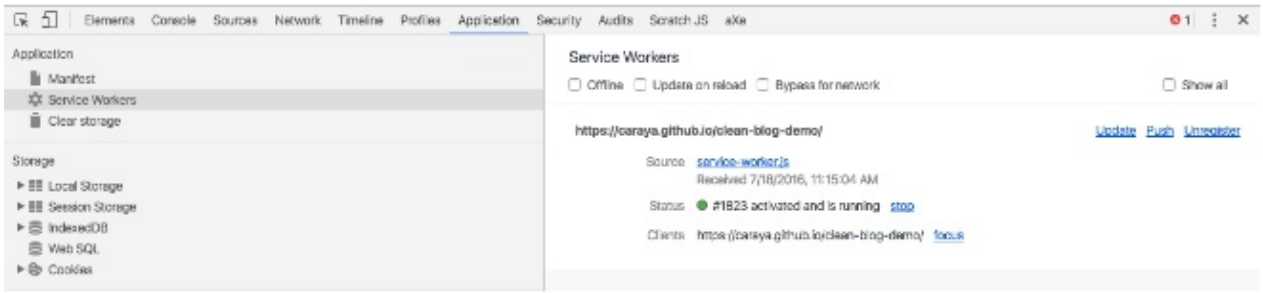
https://developers.google.com/web/tools/chrome-devtools/?hl=en

Opera

To activate Devtools in Opera first show the developer menu. From the **View** menu click **Show Developer Menu** 

Then from the **Developer** menu open the Developer Tools by clicking on **Developer Tools** or using ***F12 or Ctrl + Shift+ I* on Windows** Or ***Cmd + Opt + I on Mac***





Once in Devtools you can check the status of your service worker under application. When you click the application tab you will see the current page's service worker in the right side of the screen along with some developer helpers to make working with the service worker easier.

There are three checkboxes:

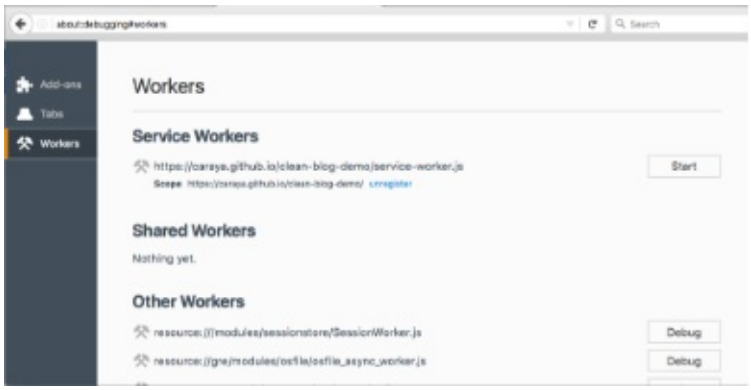
- **Offline**: forces the browser offline, simulating no network connectivity
- **Update on Reload**: reloads the service worker every time the browser reloads, picking up all changes during development
- **Bypass for network**: Ignores the service worker and fetches all resources from the network

Firefox

Firefox service worker debugging tools work slightly different that Chrome and Opera. Rather than opening the Dev Tools window, type the following in your location bar:

about:debugging

This will open the debugger window. It is a global tool; from here you can debug add-ons, tabs and their content as well as workers (web workers, shared workers and service workers).



An active service worker looks like the image below:

From here we can unregister the service worker, push messages to active clients and debug it in Firefox's Javascript debugger.