Table of Contents

Introduction	1.1
Setting Up the Labs	1.2
Lab: Responsive Design	1.3
Lab: Responsive Images	1.4
Lab: Scripting the Service Worker	1.5
Lab: Offline Quickstart	1.6
Lab: Promises	1.7
Lab: Fetch API	1.8
Lab: Caching Files with Service Worker	1.9
Lab: IndexedDB	1.10
Lab: Auditing with Lighthouse	1.11
Lab: Gulp Setup	1.12
Lab: Workbox	1.13
Lab: Migrating to Workbox from sw-precache and sw-toolbox	1.14
Lab: Integrating Web Push	1.15
Lab: Integrating Analytics	1.16
E-Commerce Lab 1: Create a Service Worker	1.17
E-Commerce Lab 2: Add to Homescreen	1.18
E-Commerce Lab 3: PaymentRequest API	1.19
Tools for PWA Developers	1.20
FAQ and Debugging	1.21

Progressive Web Apps ILT - Codelabs

This instructor-led training course for progressive web apps (PWAs) was developed by Google Developer Training.

Progressive web apps (PWAs) is the term for the open and cross-browser technology that provides better user experiences on the mobile web. Google is supporting PWAs to help developers provide native-app qualities in web applications that are reliable, fast, and engaging. The goal of PWAs is to build the core of a responsive web app and add technologies incrementally when these technologies enhance the experience. That’s the *progressive* in Progressive Web Apps!

Let's get started!

Setting Up the Labs

Use these instructions to install the lab repository on your computer prior to starting the labs. You must also install Node and set up a local Node server.

Before you begin

Development OS

These labs are intended to be used on systems running Windows 7 and later, macOS, and Linux.

Browsers

Part of the value of progressive web apps is in their ability to scale functionality to the user's browser and computing device (progressive enhancements). Although individual labs may require a specific level of support for progressive web apps, we recommend trying out the labs on multiple browsers (where feasible) so that you get a sense of how different users might experience the app.

Node

We recommend installing the latest long term support (LTS) version of Node (currently v6.9.2, which includes npm 3.10.9) rather than the most current version with the latest features.

If you have an existing version of Node installed that you would like to keep, you can install a Node version manager (for macOS and Linux platforms and Windows). This tool (nvm) lets you install multiple versions of Node, and easily switch between them. If you have issues with a specific version of Node, you can switch to another version with a single command.

Global settings

Although not a hard requirement, for general development it can be useful to disable the HTTP cache.

Set up Node and install the lab repository

Install Node and run a local Node server (you may need administrator privileges to do this).

1. Install Node by running one of the following commands from the command line:
   - If you have installed Node Version Manager (for macOS, Linux, or Windows):

     ```
     nvm install node <version>
     ```

     For example:

     ```
     nvm install node 6.11.2
     ```

     For the Windows version you can specify whether to install the 32-bit or 64-bit binaries. For example:

     ```
     nvm install node 6.11.2 64
     ```

   - If you did not install nvm, download and install Node from the Node.js website.

   This also installs Node's package manager, npm.

2. Check that Node and npm are both installed by running the following commands from the command line:

   ```
   node -v
   ```

   ```
   npm -v
   ```

   If both commands return a version number, then the installations were successful.

3. Install a simple Node server with the following command:

   ```
   npm install http-server -g
   ```

4. Clone the course repository with Git using the following command:

   ```
   git clone https://github.com/google-developer-training/pwa-training-labs.git
   ```

   Note: If you do not use Git, then download the repo from GitHub.

5. Navigate into the cloned repo:

```
cd pwa-training-labs
```

Note that some projects in the download contain folders that correspond to checkpoints in the lab (in case you get stuck during the labs, you can refer back to the checkpoints to get back on track).

6. From the **pwa-training-labs** directory, run the server with the following:

Note: If this command blocks your command-line, open a new command line window.

```
http-server -p 8080 -a localhost -c 0
```

Remember to restart the server if you shut down your computer, or end the process using `Ctrl-c`.

Explanation

Node packages are used throughout the labs. Npm will allow easy package installation. The `http-server` server lets you test your code on **localhost:8080**.

Lab: Responsive Design

Contents

Overview

1. Get set up

2. Test the page

3. Set the visual viewport

4. Use media queries

5. Using Flexbox

6. Using Flexbox as a progressive enhancement

Congratulations!

Overview

This lab shows you how to style your content to make it responsive.

What you will learn

- How to style your app so that it works well in multiple form factors
- How to use Flexbox to easily organize your content into columns
- How to use media queries to reorganize your content based on screen size

What you should know

- Basic HTML and CSS

What you will need

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

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open your browser and navigate to **localhost:8080/responsive-design-lab/app**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **responsive-design-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **index.html** is the main HTML page for our sample site/application
- **modernizr-custom.js** is a feature detection tool that simplifies testing for Flexbox support
- **styles/main.css** is the cascading style sheet for the sample site

2. Test the page

Return to the app in the browser. Try shrinking the window width to below 500px and notice that the content doesn't respond well.

Open developer tools and enable responsive design or device mode in your browser. This mode simulates the behavior of your app on a mobile device. Notice that the page is zoomed out to fit the fixed-width content on the screen. This is not a good experience because the content will likely be too small for most users, forcing them to zoom and pan.

3. Set the visual viewport

Replace TODO 3 in **index.html** with the following tag:

index.html

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Save the file. Refresh the page in the browser and check the page in device mode. Notice the page is no longer zoomed out and the scale of the content matches the scale on a desktop device. If the content behaves unexpectedly in the device emulator, toggle in and out of device mode to reset it.

Warning: Device emulation gives you a close approximation as to how your site will look on a mobile device, but to get the full picture you should always test your site on real devices. You can learn more about debugging Android devices on Chrome and Firefox.

Explanation

A meta viewport tag gives the browser instructions on how to control the page's dimensions and scaling. The `width` property controls the size of the viewport. It can be set to a specific number of pixels (for example, `width=500`) or to the special value `device-width,` which is the width of the screen in CSS pixels at a scale of 100%. (There are corresponding `height` and `device-height` values, which can be useful for pages with elements that change size or position based on the viewport height.)

The initial-scale property controls the zoom level when the page is first loaded. Setting initial scale improves the experience, but the content still overflows past the edge of the screen. We'll fix this in the next step.

For more information

- Set the viewport - Responsive Web Design Basics
- Using the viewport meta tag to control layout on mobile browsers - MDN

4. Use media queries

Replace TODO 4 in **styles/main.css** with the following code:

main.css

```
@media screen and (max-width: 48rem) {
  .container .col {
    width: 95%;
  }
}
```

Save the file. Disable device mode in the browser and refresh the page. Try shrinking the window width. Notice that the content switches to a single column layout at the specified width. Re-enable device mode and observe that the content responds to fit the device width.

Explanation

To make sure that the text is readable we use a media query when the browser's width becomes 48rem (768 pixels at browser's default font size or 48 times the default font size in the user's browser). See When to use Em vs Rem for a good explanation of why rem is a good choice for relative units. When the media query is triggered we change the layout from three columns to one column by changing the `width` of each of the three `div`s to fill the page.

5. Using Flexbox

The Flexible Box Layout Module (Flexbox) is a useful and easy-to-use tool for making your content responsive. Flexbox lets us accomplish the same result as in the previous steps, but it takes care of any spacing calculations for us and provides a bunch of ready-to-use CSS properties for structuring content.

5.1 Comment out existing rules in CSS

Comment out all of the rules in **styles/main.css** by wrapping them in `/*` and `*/`. We will make these our fallback rules for when Flexbox is not supported in the Flexbox as progressive enhancement section.

5.2 Add Flexbox layout

Replace TODO 5.2 in **styles/main.css** with the following code:

main.css

```
.container {
  display: -webkit-box;  /* OLD - iOS 6-, Safari 3.1-6 */
  display: -ms-flexbox;  /* TWEENER - IE 10 */
  display: flex;         /* NEW, Spec - Firefox, Chrome, Opera */
  background: #eee;
  overflow: auto;
}

.container .col {
  flex: 1;
  padding: 1rem;
}
```

Save the code and refresh **index.html** in your browser. Disable device mode in the browser and refresh the page. If you make your browser window narrower, the columns grow thinner until only one of them remains visible. We'll fix this with media queries in the next exercise.

Explanation

The first rule defines the `container` `div` as the flex container. This enables a flex context for all its direct children. We are mixing old and new syntax for including Flexbox to get broader support (see **For more information** for details).

The second rule uses the `.col` class to create our equal width flex children. Setting the first argument of the `flex` property to `1` for all `div`s with class `col` divides the remaining space evenly between them. This is more convenient than calculating and setting the relative width ourselves.

For more information

- A Complete Guide to Flexbox - CSS Tricks
- CSS Flexible Box Layout Module Level 1 - W3C
- What CSS to prefix?
- Using Flexbox - CSS Tricks

5.3 Optional: Set different relative widths

Use the nth-child pseudo-class to set the relative widths of the first two columns to 1 and the third to 1.5. You must use the `flex` property to set the relative widths for each column. For example, the selector for the first column would look like this:

```
.container .col:nth-child(1)
```

5.4 Use media queries with Flexbox

Replace TODO 5.4 in **styles/main.css** with the code below:

main.css

```
@media screen and (max-width: 48rem) {
  .container {
    display: -webkit-box;
    display: -ms-flexbox;
    display: flex;
    flex-flow: column;
  }
}
```

Save the code and refresh **index.html** in your browser. Now if you shrink the browser width, the content reorganizes into one column.

Explanation

When the media query is triggered we change the layout from three-column to one-column by setting the `flex-flow` property to `column`. This accomplishes the same result as the media query we added in step 5. Flexbox provides lots of other properties like `flex-flow` that let you easily structure, re-order, and justify your content so that it responds well in any context.

6. Using Flexbox as a progressive enhancement

As Flexbox is a relatively new technology, we should include fallbacks in our CSS.

6.1 Add Modernizr

Modernizr is a feature detection tool that simplifies testing for Flexbox support.

Replace TODO 6.1 in **index.html** with the code to include the custom Modernizr build:

index.html

```
<script src="modernizr-custom.js"></script>
```

Explanation

We include a Modernizr build at the top of **index.html**, which tests for Flexbox support. This runs the test on page-load and appends the class `flexbox` to the `<html>` element if the browser supports Flexbox. Otherwise, it appends a `no-flexbox` class to the `<html>` element. In the next section we add these classes to the CSS.

Note: If we were using the `flex-wrap` property of Flexbox, we would need to add a separate Modernizr detector just for this feature. Older versions of some browsers partially support Flexbox, and do not include this feature.

6.2 Use Flexbox progressively

Let's use the `flexbox` and `no-flexbox` classes in the CSS to provide fallback rules when Flexbox is not supported.

Now in **styles/main.css**, add `.no-flexbox` in front of each rule that we commented out:

main.css

```
.no-flexbox .container {
  background: #eee;
  overflow: auto;
}

.no-flexbox .container .col {
    width: 27%;
    padding: 30px 3.15% 0;
    float: left;
}

@media screen and (max-width: 48rem) {
  .no-flexbox .container .col {
    width: 95%;
  }
}
```

In the same file, add `.flexbox` in front of the rest of the rules:

main.css

```
.flexbox .container {
  display: -webkit-box;
  display: -ms-flexbox;
  display: flex;
  background: #eee;
  overflow: auto;
}

.flexbox .container .col {
  flex: 1;
  padding: 1rem;
}

@media screen and (max-width: 48rem) {
    .flexbox .container {
        display: -webkit-box;
        display: -ms-flexbox;
        display: flex;
        flex-flow: column;
    }
}
```

Remember to add `.flexbox` to the rules for the individual columns if you completed the optional step 5.3.

Save the code and refresh **index.html** in the browser. The page should look the same as before, but now it works well in any browser on any device. If you have a browser that doesn't support Flexbox, you can test the fallback rules by opening **index.html** in that browser.

For more information

- Migrating to Flexbox - Cutting the Mustard
- Modernizr Documentation

Congratulations!

You have learned to style your content to make it responsive. Using media queries, you can change the layout of your content based on the window or screen size of the user's device.

What we've covered

- Setting the visual viewport
- Flexbox

- Media queries

Resources

Learn more about the basics of responsive design

- Responsive Web Design Basics - Set the viewport
- A tale of two viewports

Learn more about Flexbox as a progressive enhancement

- Progressive Enhancement: Start Using CSS Without Breaking Older Browsers
- Migrating to Flexbox by Cutting the Mustard
- Modernizr

Learn about libraries for responsive CSS

- Bootstrap
- Sass
- Less
- Material Design

Learn more about using media queries

- Using Media Queries

Lab: Responsive Images

Contents

Overview

1. Get set up

2. Set the relative width

3. Using the srcset attribute

4. Using the sizes attribute

5. Using media queries

6. Optional: Use the picture and source elements

Congratulations!

Overview

This lab shows you how to make images on your web page look good on all devices.

What you will learn

- How to make your images responsive so that they are sized appropriately for multiple form factors
- How to use `srcset` and `sizes` to display the right image for the viewport width
- How to use `<picture>` and `source` in combination with media queries so that images on the page automatically respond as the window is resized

What you should know

- Basic HTML and CSS

What you will need

- Text editor

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open your browser and navigate to **localhost:8080/responsive-images-lab/app**.

Note: If you have installed a service worker on localhost before, unregister it so that it doesn't interfere with the lab.
If you have a text editor that lets you open a project, open the **responsive-images-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **images** folder contains sample images, each with several versions at different resolutions
- **index.html** is the main HTML page for our sample site/application
- **styles/main.css** is the cascading style sheet for the sample site

2. Set the relative width

Before making the images responsive, let's make sure they won't overflow the screen.

Replace TODO 2 in **styles/main.css** with the following code:

main.css

```
img {
  max-width: 100%;
}
```

Save the code and refresh the page in your browser. Try resizing the window. The image widths should stay entirely within the window.

Explanation

The value in `max-width` represents a percentage of the containing element, in this case the `<article>` element.

Note: You could also specify the `max-width` in terms of the viewport width using `vw` units

(for example, 100vw). In this case we are using a percentage value to keep the images the same width as the text.

3. Using the srcset attribute

The goal is to get the browser to fetch the version of the image with the smallest dimensions that is still bigger than the final display size of the image. `srcset` lets us list a set of images at different resolutions for the browser to choose from when fetching the image. The browser's choice depends on the viewport dimensions, the image size relative to the viewport, the pixel density of the user's device, and the source file's dimensions.

3.1 Add a srcset to an image

To complete TODO 3.1 in **index.html**, add the following `srcset` attribute to the `<img>` element containing the SFO image:

index.html

```
srcset="images/sfo-1600_large.jpg, images/sfo-1000_large.jpg, images/sfo-800_medium.jp
g, images/sfo-500_small.jpg"
```

Save the code and refresh the page in the browser. Open your browser's Developer Tools and look at the network requests. Try refreshing the page at different window sizes. You should see that the browser is fetching **images/sfo-1600_large.jpg** no matter the window size.

Explanation

In the **images** folder there are several versions of the SFO image, each at different resolutions. We list these in the `srcset` attribute to give the browser the option to choose which file to use. However, the browser has no way of determining the file sizes before it loads them, so it always chooses the first image in the list.

3.2 Add width descriptors to the srcset

To load the correct image size based on the viewport width we need to tell the browser how big each file is before it fetches them.

To complete TODO 3.2 in **index.html**, add width descriptors to the SFO `<img>` element:

index.html

```
srcset="images/sfo-1600_large.jpg 1600w, images/sfo-1000_large.jpg 1000w, images/sfo-8
00_medium.jpg 800w, images/sfo-500_small.jpg 500w"
```

Save the code and refresh the page in the browser. Refresh the page at various window sizes and check the network requests to see which version of the image is fetched at each size. On a 1x display, the browser fetches **sfo-500_small.jpg** when the window is narrower than 500px, **sfo-800_medium.jpg** when it is narrower than 800px, and so forth.

Note: In Chrome, with **DevTools** open, the browser window dimensions appear as it is being resized (see the image below). This feature will be very useful throughout this codelab.

Explanation

By adding a width descriptor to each file in the `srcset`, we are telling the browser the width of each image in pixels *before* it fetches the image. The browser can then use these widths to decide which image to fetch based on its window size. It fetches the image with the smallest width that is still larger than the viewport width.

Note: You can also optionally specify a pixel density instead of a width. However, you cannot specify both pixel densities and widths in the same `srcset` attribute. We explore using pixel densities in a later section.

4. Using the sizes attribute

4.1 Display an image at half the width of the viewport (50vw)

Replace TODO 4.1 in **styles/main.css** with the following code:

styles/main.css

```
img#sfo {
  transition: width 0.5s;
  max-width: 50vw;
}
```

Save the code and refresh the page in the browser. Try refreshing the page at various window sizes and check the network requests at each size. The browser is fetching the same sized images as before.

Explanation

Because the CSS is parsed after the HTML at runtime, the browser has no way to know what the final display size of the image will be when it fetches it. Unless we tell it otherwise, the browser assumes the images will be displayed at 100% of the viewport width and fetches the images based on this. We need a way to tell the browser beforehand if the images will be displayed at a different size.

4.2 Add the sizes attribute to the image

We can give `<img>` a `sizes` attribute to tell the browser the display size of the image before it is fetched.

To complete TODO 4.2 in **index.html** add `sizes="50vw"` to the `img` element so that it looks like this:

index.html

```
<img id="sfo" src="images/sfo-500_small.jpg" srcset="images/sfo-1600_large.jpg 1600w,
images/sfo-1000_large.jpg 1000w, images/sfo-800_medium.jpg 800w, images/sfo-500_small.
jpg 500w" sizes="50vw" alt="View from aircraft window near San Francisco airport">
```

Save the code and refresh the page in the browser. Refresh the page at various window sizes and check the network requests each time. You should see that for the same approximate window sizes you used to test the previous step, the browser is fetching a smaller image.

Explanation

The `sizes` value matches the image's `max-width` value in the CSS. The browser now has everything it needs to choose the correct image version. The browser knows its own viewport width and the pixel density of the user's device, and we have given it the source files' dimensions (using the width descriptor) and the image sizes relative to the viewport (using the `sizes` attribute).

For more information

- Srcset and sizes

5. Using media queries

5.1 Add a media query to the CSS

We can use media queries to resize images in real time based on the viewport width.

Replace TODO 5.1 in **styles/main.css** with the following code:

styles/main.css

```
@media screen and (max-width: 700px) {
  img#sfo {
    max-width: 90vw;
    width: 90vw;
  }
}
```

Save the code and refresh the page in the browser. Shrink the window to less than 700px (in Chrome, the viewport dimensions are shown on the screen if **DevTools** is open). The image should resize to fill 90% of the window width.

Explanation

The media query tests the viewport width of the screen, and applies the CSS if the viewport is less than 700px wide.

For more information

- @media

5.2 Add the media query to the sizes attribute

We can tell the browser about the media query in the `sizes` attribute so that it fetches the correct image when the image changes size.

To complete TODO 5.2 in **index.html**, update the `sizes` attribute in the SFO image:

index.html

```
sizes="(max-width: 700px) 90vw, 50vw"
```

Save the code and refresh the page in the browser. Resize the browser window so that it is 600px wide. On a 1x display, the browser should fetch **sfo-800_medium.jpg**.

6. Optional: Use the picture and source elements

We can use the `<picture>` element and the `<source>` element, in combination with media queries, to change the image source as the window is resized.

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

index.html

```
<figure>
    <picture>
    <source media="(min-width: 750px)"
            srcset="images/horses-1600_large_2x.jpg 2x,
                    images/horses-800_large_1x.jpg" />
    <source media="(min-width: 500px)"
            srcset="images/horses_medium.jpg" />
    <img src="images/horses_small.jpg" alt="Horses in Hawaii">
    </picture>
    <figcaption>Horses in Hawaii</figcaption>
</figure>
```

Save the code and refresh the page in the browser. Try resizing the browser window. You should see the image change at 750px and 500px.

Explanation

The `<picture>` element lets us define multiple source files using the `<source>` tag. This is different than simply using an `<img>` tag with the `srcset` attribute because the source tag lets us add things like media queries to each set of sources. Instead of giving the browser the image sizes and letting it decide which files to use, we can define the images to use at each window size.

We have included several versions of the sample image, each at different resolutions and cropped to make the focus of the image visible at smaller sizes. In the code above, at larger than 750px, the browser fetches either **horses-1600_large_2x.jpg** (if the device has a 2x display) or **horses-800_large_1x.jpg**. If the window's width is less than 750px but greater than 500px, the browser fetches **horses_medium.jpg**. At less than 500px the browser fetches the fallback image, **horses_small.jpg**.

Note: If the user's browser doesn't support the `<picture>` element, it fetches whatever is in the `<img>` element. The `<picture>` element is just used to specify multiple sources for the `<img>` element contained in it. The `<img>` element is what displays the image.

For more information

- picture element - MDN
- source element - MDN

Congratulations!

You have learned how to make images on your web page look good on all devices!

Resources

Learn about automating the process

- Gulp responsive images (NPM) - requires libvips on Mac
- Gulp responsive images (GitHub) - requires graphicsmagick on all platforms
- Responsive Image Breakpoints Generator v2.0

Learn more about srcset and sizes

- Srcset and sizes
- Responsive Images: If you're just changing resolutions, use srcset.

Learn more about art direction

- Use Cases and Requirements for Standardizing Responsive Images

Lab: Scripting the Service Worker

Contents

Overview

1. Get set up

2. Register the service worker

3. Listening for life cycle events

4. Intercept network requests

5. Optional: Exploring service worker scope

Congratulations!

Concepts: Introduction to Service Worker

Overview

This lab walks you through creating a simple service worker.

What you will learn

- Create a basic service worker script, install it, and do simple debugging

What you should know

- Basic JavaScript and HTML
- Concepts and basic syntax of ES2015 Promises
- Concept of an Immediately Invoked Function Expression (IIFE)
- 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

- A text editor

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open your browser and navigate to **localhost:8080/service-worker-lab/app**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **service-worker-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **other.html**, **js/other.js**, **below/another.html**, and **js/another.js** are sample resources that we use to experiment
- **index.html** is the main HTML page for our sample site/application
- **index.css** is the cascading stylesheet for **index.html**
- **service-worker.js** is the JavaScript file that is used to create our service worker
- **styles** folder contains the cascading stylesheets for this lab
- **test** folder contains files for testing your progress

2. Register the service worker

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

Note: We are using an Immediately Invoked Function Expression inside the service worker. This is just a best practice for avoiding namespace pollution; it is not related to the Service Worker API.
Open **index.html** in your text editor.

Replace TODO 2 with the following code:

index.html

```
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.

In your browser, navigate to **test-registered.html** (**app/test/test-registered.html**) to confirm that you have registered the service worker. This is a unit test. Passed tests are blue and failed tests are red. If you've done everything correctly so far, this test should be blue. Close the test page when you are done with it.

Note: Be sure to open the test page using the localhost address so that it opens from the server and not directly from the file system.
Optional: Open the site on an unsupported browser and verify that the support check conditional works.

Explanation

Service workers must be registered. 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 our code, 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, the promise will reject.

3. Listening for life cycle events

Changes in the service worker's status trigger events in the service worker.

3.1 Add event listeners

Open **service-worker.js** in your text editor.

Replace TODO 3.1 with the following code:

service-worker.js

```
self.addEventListener('install', function(event) {
  console.log('Service worker installing...');
  // TODO 3.4: Skip waiting
});

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

Save the file. Close **app/test/test-registered.html** page if you have not already. Manually unregister the service worker and refresh the page to install and activate the updated service worker. The console log should indicate that the new service worker was registered, installed, and activated.

Note: All pages associated with the service worker must be closed before an updated service worker can take over.
Note: The registration log may appear out of order with the other logs (installation and activation). The service worker runs concurrently with the page, so we can't guarantee the order of the logs (the registration log comes from the page, while the installation and activation logs come from the service worker). Installation, activation, and other service worker events occur in a defined order inside the service worker, however, and should always appear in the expected order.

Explanation

The 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 there is no registered 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 emits an `activate` event when it takes control of the page. We log a message here, but this event is often used to update caches.

Only one service worker can be active at a time for a given scope (see Exploring service worker scope), so a newly installed 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 unregistered the existing service worker, the new service worker was activated immediately.

Note: Simply refreshing the page is not sufficient to transfer control to a new service worker, because the new page will be requested before the the current page is unloaded, and there won't be a time when the old service worker is not in use.
Note: You can also manually activate a new service worker using some browsers' developer tools and programmatically with `skipWaiting()`, which we discuss in section 3.4.

3.2 Re-register the existing service worker

Reload the page. Notice how the events change.

Now close and reopen the page (remember to close all pages associated with the service worker). Observe the logged events.

Explanation

After initial installation and activation, re-registering an existing worker does not re-install or re-activate the service worker. Service workers also persist across browsing sessions.

3.3 Update the service worker

Replace TODO 3.3 in **service-worker.js** with the following comment:

service-worker.js

```
// I'm a new service worker
```

Save the file and refresh the page. Notice that the new service worker installs but does not activate.

Navigate to **test-waiting.html** (**app/test/test-waiting.html**) to confirm that the new service worker is installed but not activated. The test should be passing (blue).

Close all pages associated with the service worker (including the **app/test/test-waiting.html** page). Reopen the **app/** page. The console log should indicate that the new service worker has now activated.

Note: If you are getting unexpected results, make sure your HTTP cache is disabled in developer tools.

Explanation

The browser detects a byte difference between the new and existing service worker file (because of the added comment), so the new service worker is installed. Since only one service worker can be active at a time (for a given scope), even though the new service worker is installed, it isn't activated until the existing service worker is no longer in use. By closing all pages under the old service worker's control, we are able to activate the new service worker.

3.4 Skipping the waiting phase

It is possible for a new service worker to activate immediately, even if an existing service worker is present, by skipping the waiting phase.

Replace TODO 3.4 in **service-worker.js** with the following code:

service-worker.js

```
self.skipWaiting();
```

Save the file and refresh the page. Notice that the new service worker installs and activates immediately, even though a previous service worker was in control.

Explanation

The `skipWaiting()` method allows a service worker to activate as soon as it finishes installation. The install event listener is a common place to put the `skipWaiting()` call, but it can be called anywhere during or before the waiting phase. See this documentation for more on when and how to use `skipWaiting()`. For the rest of the lab, we can now test new service worker code without manually unregistering the service worker.

For more information

- Service worker lifecycle

4. Intercept network requests

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

Replace TODO 4 in **service-worker.js** with:

service-worker.js

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

Save the script and refresh the page to install and activate the updated service worker.

Check the console and observe that no fetch events were logged. Refresh the page and check the console again. You should see fetch events this time for the page and its assets (like CSS).

Click the links to **Other page**, **Another page**, and **Back**.

You'll see fetch events in the console for each of the pages and their assets. Do all the logs make sense?

Note: If you visit a page and do not have the HTTP 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 receives a fetch event for every HTTP request made by the browser. The fetch event object contains the request. Listening for fetch events in the service worker is similar to listening to click events in the DOM. In our code, when a fetch event occurs, we log the requested URL to the console (in practice we could also create and return our own custom response with arbitrary resources).

Why didn't any fetch events log on the first refresh? By default, fetch events from a page won't go through a service worker unless the page request itself went through a service worker. This ensures consistency in your site; if a page loads without the service worker, so do its subresources.

For more information

- Fetch Event - MDN
- Using Fetch - MDN
- Introduction to Fetch - Google Developer

Solution code

To get a copy of the working code, navigate to the **04-intercepting-network-requests** folder.

5. Optional: Exploring service worker scope

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

5.1 Find the scope

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

index.html

```
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:8080/service-worker-lab/app/**).

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. So a service worker in the root directory of an app controls requests from all files in the app.

5.2 Move the service worker

Unregister the current service worker.

Then move **service-worker.js** into the **app/below** directory and update the service worker URL in the registration code. Unregister the service worker and refresh the page.

The console shows that the scope of the service worker is now **localhost:8080/service-worker-lab/app/below/**.

Navigate to **test-scoped.html** (**app/test/test-scoped.html**) to confirm that that service worker is registered in **app/below/**. If you've done everything correctly, you shouldn't see any red errors. Close the test page when you are done with it.

Back on the main page, click **Other page**, **Another page** and **Back**. Which fetch requests are being logged? Which aren't?

Explanation

The service worker's default scope is the path to the service worker file. Since the service worker file is now in **app/below/**, that is its scope. The console is now only logging fetch events for **another.html**, **another.css**, and **another.js**, because these are the only resources within the service worker's scope (**app/below/**).

5.3 Set an arbitrary scope

Unregister the current service worker again.

Move the service worker back out into the project root directory (**app**) and update the service worker URL in the registration code.

Use the reference on MDN to set the scope of the service worker to the **app/below/** directory using the optional parameter in `register()`. Unregister the service worker and refresh the page. Click **Other page**, **Another page** and **Back**.

Again the console shows that the scope of the service worker is now **localhost:8080/service-worker-lab/app/below**, and logs fetch events only for **another.html**, **another.css**, and **another.js**.

Navigate to **test-scoped.html** again to confirm that the service worker is registered in **app/below/**.

Explanation

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

index.html

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

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

Note: You cannot set an arbitrary scope that is above the service worker's actual location.

For more information

- Service worker registration object
- The register() method
- Service worker scope

Solution code

To get a copy of the working code, navigate to the **solution** folder.

Congratulations!

You now have a simple service worker up and running.

Lab: Offline Quickstart

Contents

Overview

1. Get set up

2. Taking the app offline

Congratulations!

Concepts: Offline Quickstart

Overview

This lab shows you how to add offline capabilities to an application using service workers.

What you will learn

- How to add offline capabilities to an application

What you should know

- Basic HTML, CSS, and JavaScript
- Familiarity with ES2015 Promises

What you will need

- Computer with terminal/shell access
- Connection to the internet
- A browser that supports service workers
- A text editor

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open your browser and navigate to **localhost:8080/offline-quickstart-lab/app**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **offline-quickstart-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **images** folder contains sample images
- **styles/main.css** is the main cascading stylesheet for the app
- **index.html** is the main HTML page for our sample site/application
- **service-worker.js** is the service worker file (currently empty)

2. Taking the app offline

Let's create a service worker to add offline functionality to the app.

2.1 Cache static assets on install

Replace the TODO 2.1 comment in **service-worker.js** with the following code:

service-worker.js

```
var CACHE_NAME = 'static-cache';

var urlsToCache = [
  '.',
  'index.html',
  'styles/main.css'
];

self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME)
    .then(function(cache) {
      return cache.addAll(urlsToCache);
    })
  );
});
```

Save the file.

Explanation

This code starts by defining a cache name, and a list of URLs to be cached. An install event listener is then added to the service worker. When the service worker installs, it opens a cache and stores the app's static assets. Now these assets are available for quick loading from the cache, without a network request.

Note that `.` is also cached. This represents the current directory, in this case, **app/**. We do this because the browser attempts to fetch **app/** first before fetching **index.html**. When the app is offline, this results in a 404 error if we have not cached **app/**. They should both be cached to be safe.

Note: Don't worry if you don't understand all of this code; this lab is meant as an overview. The `event.waitUntil` code can be particularly confusing. This operation simply tells the browser not to preemptively terminate the service worker before the asynchronous operations inside of it have completed.

2.2 Fetch from the cache

Replace TODO 2.2 in **service-worker.js** with the following code:

service-worker.js

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

function fetchAndCache(url) {
  return fetch(url)
  .then(function(response) {
    // Check if we received a valid response
    if (!response.ok) {
      throw Error(response.statusText);
    }
    return caches.open(CACHE_NAME)
    .then(function(cache) {
      cache.put(url, response.clone());
      return response;
    });
  })
  .catch(function(error) {
    console.log('Request failed:', error);
    // You could return a custom offline 404 page here
  });
}
```

Save the script.

Explanation

This code adds a fetch event listener to the service worker. When a resource is requested, the service worker intercepts the request and a fetch event is fired. The code then does the following:

- Tries to match the request with the content of the cache, and if the resource is in the cache, then returns it.
- If the resource is not in the cache, attempts to get the resource from the network using fetch.
- If the response is invalid, throws an error and logs a message to the console (`catch`).
- If the response is valid, creates a copy of the response (`clone`), stores it in the cache, and then returns the original response.

Note: We `clone` the response because the request is a stream that can only be consumed once. Because we want to put it in the cache and serve it to the user, we need to clone a copy. See Jake Archibald's What happens when you read a response article for a more in-

depth explanation.

2.3 Register the service worker

Replace TODO 2.3 in **index.html** with the following code:

index.html

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('service-worker.js')
  .then(function(registration) {
    console.log('Registered:', registration);
  })
  .catch(function(error) {
    console.log('Registration failed: ', error);
  });
}
```

Save the file.

Explanation

This code first checks that service worker is supported by the browser. If it is, the service worker that we just wrote is registered, beginning the installation process.

2.4 Test the app offline

Now our app has offline functionality. Save all files and refresh the **app/** in the browser. You can check the cache and see that the HTML and CSS are cached from the service worker installation event.

Refresh the page again. This fetches all of the page's assets, and the fetch listener caches any asset that isn't already cached.

Stop the server (use `Ctrl+c` if your server is running from the command line) or switch the browser to offline mode to simulate going offline. Then refresh the page. The page should load normally!

Note: You may see an error when the page tries to fetch the service worker script. This is because the browser attempts to re-fetch the service worker file for every navigation request. If offline, the attempt fails (causing an error log). However, the browser should default to the installed service worker and work as expected.

Explanation

When our app opens for the first time, the service worker is registered, installed, and activated. During installation, the app caches the most critical static assets (the main HTML and CSS). On future loads, each time a resource is requested the service worker intercepts the request, and checks the cache for the resource before going to the network. If the resource isn't in the cache, the service worker fetches it from the network and caches a copy of the response. Since we refreshed the page and fetched all of its assets, everything needed for the app is in the cache and it can now open without the network.

Note: You might be thinking, why didn't we just cache everything on install? Or, why did we cache anything on install, if all fetched resources are cached? This lab is intended as an overview of how you can bring offline functionality to an app. In practice, there are a variety of caching strategies and tools that let you customize your app's offline experience. Check out the Offline Cookbook for more info.

Solution code

To get a copy of the working code, navigate to the **solution** folder.

Congratulations!

You now know the basics of adding offline functionality to an app.

Lab: Promises

Contents

Overview

1. Get set up

2. Using promises

3. Chaining promises

4. Optional: Using Promise.all and Promise.race

Congratulations!

Concepts: Working with Promises

Overview

This lab teaches you how to use JavaScript Promises.

What you will learn

- How to create promises
- How to chain promises together
- How to handle errors in promises
- How to use Promise.all and Promise.race

What you should know

- Basic JavaScript and HTML
- The concept of an Immediately Invoked Function Expression (IIFE)
- How to enable the developer console

What you will need

- A browser that supports Promises and Fetch
- A text editor
- Computer with terminal/shell access

- Connection to the internet

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open your browser and navigate to **localhost:8080/promises-lab/app**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **promises-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **flags/chile.png**, **flags/peru.png**, **flags/spain.png** - sample resources that we use to experiment
- **js/main.js** is the main JavaScript file for the app
- **test/test.html** is a file for testing your progress
- **index.html** is the main HTML page for our sample site/application

2. Using promises

This step uses Promises to handle asynchronous code in JavaScript.

2.1 Create a promise

Let's start by creating a simple promise.

Complete the `getImageName` function by replacing TODO 2.1 in **js/main.js** with the following code:

main.js

```
country = country.toLowerCase();
var promiseOfImageName = new Promise(function(resolve, reject) {
  setTimeout(function() {
    if (country === 'spain' || country === 'chile' || country === 'peru') {
      resolve(country + '.png');
    } else {
      reject(Error('Didn\'t receive a valid country name!'));
    }
  }, 1000);
});
console.log(promiseOfImageName);
return promiseOfImageName;
```

Save the script and refresh the page.

Enter "Spain" into the app's **Country Name** field. Then, click **Get Image Name**. You should see a Promise object logged in the console.

Now enter "Hello World" into the **Country Name** field and click **Get Image Name**. You should see another Promise object logged in the console, followed by an error.

Note: Navigate to **app/test/test.html** in the browser to check your function implementations. Functions that are incorrectly implemented or unimplemented show red errors. Be sure to open the test page using the localhost address so that it opens from the server and not directly from the file system.

Explanation

The `getImageName` function creates a promise. A promise represents a value that might be available now, in the future, or never. In effect, a promise lets an asynchronous function such as `getImageName` (the `setTimeout` method is used to make `getImageName` asynchronous) return a value much like a synchronous function. Rather than returning the final value (in this case, "Spain.png"), `getImageName` returns a promise of a future value (this is what you see in the console log). Promise construction typically looks like this example at developers.google.com:

main.js

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then...

  if (/<em> everything turned out fine </em>/) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

Depending on the outcome of an asynchronous operation, a promise can either resolve with a value or reject with an error. In the `getImageName` function, the `promiseOfImageName` promise either resolves with an image filename, or rejects with a custom error signifying that the function input was invalid.

Optional: Complete the `isSpain` function so that it takes a string as input, and returns a new promise that resolves if the function input is "Spain", and rejects otherwise. You can verify that you implemented `isSpain` correctly by navigating to **app/test/test.html** and checking the `isSpain` test. Note that this exercise is optional and is not used in the app.

2.2. Use the promise

This section uses the promise we just created.

Replace TODO 2.2 inside the `flagChain` function in **js/main.js** with the following code:

main.js

```
return getImageName(country)
.then(logSuccess, logError);
```

Save the script and refresh the page.

Enter "Spain" into the app's **Country Name** field again. Now click **Flag Chain**. In addition to the promise object, "Spain.png" should now be logged.

Now enter "Hello World" into the **Country Name** text input and click **Flag Chain** again. You should see another promise logged in the console, followed by a custom error message.

Explanation

The `flagChain` function returns the result of `getImageName`, which is a promise. The then method lets us implicitly pass the settled (either resolved or rejected) promise to another function. The `then` method takes two arguments in the following order:

1. The function to be called if the promise resolves.
2. The function to be called if the promise rejects.

If the first function is called, then it is implicitly passed the resolved promise value. If the second function is called, then it is implicitly passed the rejection error.

Note: We used named functions inside `then` as good practice, but we could use anonymous functions as well.

2.3 Use catch for error handling

Let's look at the `catch` method, which is a clearer alternative for error handling.

Replace the code inside the `flagChain` function with the following:

main.js

```
return getImageName(country)
.then(logSuccess)
.catch(logError);
```

Save the script and refresh the page. Repeat the experiments from section 2.2 and note that the results are identical.

Explanation

The catch method is similar to `then`, but deals only with rejected cases. It behaves like `then(undefined, onRejected)`. With this new pattern, if the promise from `getImageName` resolves, then `logSuccess` is called (and is implicitly passed the resolved promise value). If the promise from `getImageName` rejects, then `logError` is called (and implicitly passed the rejection error).

This code is not quite equivalent to the code in section 2.2, however. This new code also triggers `catch` if `logSuccess` rejects, because `logSuccess` occurs before the `catch`. This new code would actually be equivalent to the following:

main.js

```
return getImageName(country)
.then(logSuccess)
.then(undefined, logError);
```

The difference is subtle, but extremely useful. Promise rejections skip forward to the next `then` with a rejection callback (or `catch`, since they're equivalent). With `then(func1, func2)`, `func1` or `func2` will be called, never both. But with `then(func1).catch(func2)`, both will be called if `func1` rejects, as they're separate steps in the chain.

Optional: If you wrote the optional `isSpain` function in section 2.1, complete the `spainTest` function so that it takes a string as input and returns a promise using an `isSpain` call with the input string. Use `then` and `catch` such that `spainTest` returns a value of true if the `isSpain` promise resolves and false if the `isSpain` promise rejects (you can use the `returnTrue` and `returnFalse` helper functions). You can verify that you have implemented `spainTest` correctly by navigating to **app/test/test.html** and checking the `spainTest` test.

For more information

- Promise object
- Promises introduction
- Resolve
- Reject
- Then
- Catch

Solution code

To get a copy of the working code, navigate to the **02-basic-promises** folder.

3. Chaining promises

The `then` and `catch` methods also return promises, making it possible to chain promises together.

3.1 Add asynchronous steps

Replace the code in the `flagChain` function with the following:

main.js

```
return getImageName(country)
.then(fetchFlag)
.then(processFlag)
.then(appendFlag)
.catch(logError);
```

Save the script and refresh the page.

Enter "Spain" into the app's **Country Name** text input. Now click **Flag Chain**. You should see the Spanish flag display on the page.

Now enter "Hello World" into the **Country Name** text input and click **Flag Chain**. The console should show that the error is triggering `catch`.

Explanation

The updated `flagChain` function does the following:

1. As before, `getImageName` returns a promise. The promise either resolves with an image file name, or rejects with an error, depending on the function's input.
2. If the returned promise resolves, then the image file name is passed to `fetchFlag` inside the first `then`. This function requests the corresponding image file asynchronously, and returns a promise (see fetch documentation).
3. If the promise from `fetchFlag` resolves, then the resolved value (a response object) is passed to `processFlag` in the next `then`. The `processFlag` function checks if the response is ok, and throws an error if it is not. Otherwise, it processes the response with the `blob` method, which also returns a promise.
4. If the promise from `processFlag` resolves, the resolved value (a blob), is passed to the `appendFlag` function. The `appendFlag` function creates an image from the value and appends it to the DOM.

If any of the promises reject, then all subsequent `then` blocks are skipped, and `catch` executes, calling `logError`. Throwing an error in the `processFlag` function also triggers the `catch` block.

3.2 Add a recovery catch

The `flagChain` function does not add a flag to the page if an invalid country is used as input (`getImageName` rejects and execution skips to the `catch` block).

Add a `catch` to the promise chain that uses the `fallbackName` function to supply a fallback image file name to the `fetchFlag` function if an invalid country is supplied to `flagChain`. To verify this was added correctly, navigate to **app/test/test.html** and check the `flagChain`

test.

Note: This test is asynchronous and may take a few moments to complete.
Save the script and refresh the page. Enter "Hello World" in the **Country Name** field and click **Flag Chain**. Now the Chilean flag should display even though an invalid input was passed to `flagChain`.

Explanation

Because `catch` returns a promise, you can use the `catch` method inside a promise chain to *recover* from earlier failed operations.

For more information

- Fetch API

Solution code

To get a copy of the working code, navigate to the **03-chaining-promises** folder.

4. Optional: Using Promise.all and Promise.race

4.1 Promise.all

Often we want to take action only after a collection of asynchronous operations have completed successfully.

Complete the `allFlags` function such that it takes a list of promises as input. The function should use Promise.all to evaluate the list of promises. If all promises resolve successfully, then `allFlags` returns the values of the resolved promises as a list. Otherwise, `allFlags` returns `false`. To verify that you have done this correctly, navigate to **app/test/test.html** and check the `allFlags` test.

Test the function by replacing TODO 4.1 in **js/main.js** with the following code:

main.js

```
var promises = [
  getImageName('Spain'),
  getImageName('Chile'),
  getImageName('Peru')
];

allFlags(promises).then(function(result) {
  console.log(result);
});
```

Save the script and refresh the page. The console should log each promise object and show `["spain.png", "chile.png", "peru.png"]`.

Note: In this example we are using an anonymous function inside the `then` call. This is not related to `Promise.all`.
Change one of the inputs in the `getImageName` calls inside the `promises` variable to "Hello World". Save the script and refresh the page. Now the console should log `false`.

Explanation

`Promise.all` returns a promise that resolves if all of the promises passed into it resolve. If any of the passed-in promises reject, then `Promise.all` rejects with the reason of the first promise that was rejected. This is very useful for ensuring that a group of asynchronous actions complete (such as multiple images loading) before proceeding to another step.

Note: `Promise.all` would not work if the promises passed in were from `flagChain` calls because `flagChain` uses `catch` to ensure that the returned promise always resolves.
Note: Even if an input promise rejects, causing `Promise.all` to reject, the remaining input promises still settle. In other words, the remaining promises still execute, they simply are not returned by `Promise.all`.

For more information

- Promise.all documentation

4.2 Promise.race

Another promise method that you may see referenced is Promise.race.

Replace TODO 4.2 in **js/main.js** with the following code:

main.js

```
var promise1 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 500, 'one');
});

var promise2 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, 'two');
});

Promise.race([promise1, promise2])
.then(logSuccess)
.catch(logError);
```

Save the script and refresh the page. The console should show "two" logged by `logSuccess`.

Change `promise2` to reject instead of resolve. Save the script and refresh the page. Observe that "two" is logged again, but this time by `logError`.

Explanation

`Promise.race` takes a list of promises and settles as soon as the first promise in the list settles. If the first promise resolves, `Promise.race` resolves with the corresponding value, if the first promise rejects, `Promise.race` rejects with the corresponding reason.

In this example, if `promise2` resolves before `promise1` settles, the `then` block executes and logs the value of the `promise2`. If `promise2` rejects before `promise1` settles, the `catch` block executes and logs the reason for the `promise2` rejection.

Note: Because `Promise.race` rejects immediately if one of the supplied promises rejects (even if another supplied promise resolves later) `Promise.race` by itself can't be used to reliably return the first promise that resolves. See the concepts section for more details.

For more information

- Promise.race documentation

Solution code

To get a copy of the working code, navigate to the **solution** folder.

Congratulations!

You have learned the basics of JavaScript Promises!

Resources

- Promises introduction
- Promise - MDN

Lab: Fetch API

Contents

Overview

1. Get set up

2. Fetching a resource

3. Fetch an image

4. Fetch text

5. Using HEAD requests

6. Using POST requests

7. Optional: CORS and custom headers

Congratulations!

Concepts: Working with the Fetch API

Overview

This lab walks you through using the Fetch API, a simple interface for fetching resources, as an improvement over the XMLHttpRequest API.

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 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
- A text editor
- Node and npm

Note: Although the Fetch API is not currently supported in all browsers, there is a polyfill (but see the readme for important caveats).

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open your browser and navigate to **localhost:8080/fetch-api-lab/app**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **fetch-api-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **echo-servers** contains files that are used for running echo servers
- **examples** contains sample resources that we use in experimenting with fetch
- **index.html** is the main HTML page for our sample site/application
- **js/main.js** is the main JavaScript for the app, and where you will write all your code
- **test/test.html** is a file for testing your progress
- **package.json** is a configuration file for node dependencies

2. Fetching a resource

2.1 Fetch a JSON file

Open **js/main.js** in your text editor.

Replace the TODO 2.1a comment with the following code:

main.js

```
if (!('fetch' in window)) {
  console.log('Fetch API not found, try including the polyfill');
  return;
}
```

In the `fetchJSON` function, replace TODO 2.1b with the following code:

main.js

```
fetch('examples/animals.json')
.then(logResult)
.catch(logError);
```

Save the script and refresh the page. Click **Fetch JSON**. The console should log the fetch response.

Note: We are using the JavaScript module pattern in this file. This is just to help keep the code clean and allow for easy testing. It is not related to the Fetch API.
Optional: Open the site on an unsupported browser and verify that the support check conditional works.

Explanation

The code starts by checking for fetch support. If the browser doesn't support fetch, the script logs a message and fails immediately.

We pass the path for the resource we want to retrieve as a parameter to fetch, in this case **examples/animals.json**. A promise that resolves to a Response object is returned. If the promise resolves, the response is passed to the `logResult` function. If the promise rejects, the `catch` takes over and the error is passed to the `logError` function.

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

2.2 Examine response properties

Find the values of the `status`, `url`, and `ok` properties of the response for the fetch we just made. What are these values? Hint: Look in the console.

In the `fetchJSON` function we just wrote in section 2.1, replace the **examples/animals.json** resource with **examples/non-existent.json**. So the `fetchJSON` function should now look like:

main.js

```
function fetchJSON() {
  fetch('examples/non-existent.json')
  .then(logResult)
  .catch(logError);
}
```

Save the script and refresh the page. Click **Fetch JSON** again to try and fetch this new resource.

Now find the `status`, `URL`, and `ok` properties of the response for this new fetch we just made. What are these values?

The values should be different for the two files (do you understand why?). If you got any console errors, 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 404s) still resolve! A fetch promise only rejects if the request was unable to complete, so you must always check the validity of the response.

For more information

- Response objects

2.3 Check response validity

We need to update our code to check the validity of responses.

Complete the function called `validateResponse` in TODO 2.3. The function should accept a response object as input. If the response object's `ok` property is false, the function should throw an error containing `response.statusText`. If the response object's `ok` property is true, the function should simply return the response object.

You can confirm that you have written the function correctly by navigating to **app/test/test.html**. This page runs tests on some of the functions you write. If there are errors with your implementation of a function (or you haven't implemented them yet), the test displays in red. Passed tests display in blue. Refresh the **test.html** page to retest your functions.

Note: Be sure to open the test page using the localhost address so that it opens from the

server and not directly from the file system.
Once you have successfully written the function, replace `fetchJSON` with the following code:

main.js

```
function fetchJSON() {
  fetch('examples/non-existent.json')
  .then(validateResponse)
  .then(logResult)
  .catch(logError);
}
```

This is promise chaining.

Save the script and refresh the page. Click **Fetch JSON**. Now the response for **examples/non-existent.json** should trigger the `catch` block, unlike in section 2.2. Check the console to confirm this.

Now replace **examples/non-existent.json** resource in the `fetchJSON` function with the original **examples/animals.json** from section 2.1. The function should now look like:

main.js

```
function fetchJSON() {
  fetch('examples/animals.json')
  .then(validateResponse)
  .then(logResult)
  .catch(logError);
}
```

Save the script and refresh the page. Click **Fetch JSON**. You should see that the response is being logged successfully like in section 2.1.

Explanation

Now that we have added the `validateResponse` check, bad responses (like 404s) throw an error and the `catch` takes over. This prevents bad responses from propagating down the fetch chain.

2.4 Read the response

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

To complete TODO 2.4, replace the `readResponseAsJSON` function with the following code:

main.js

```
function readResponseAsJSON(response) {
  return response.json();
}
```

(You can check that you have done this correctly by navigating to **app/test/test.html**.)

Then replace the `fetchJSON` function with the following code:

main.js

```
function fetchJSON() {
  fetch('examples/animals.json') // 1
  .then(validateResponse) // 2
  .then(readResponseAsJSON) // 3
  .then(logResult) // 4
  .catch(logError);
}
```

Save the script and refresh the page. Click **Fetch JSON**. Check the console to see that the JSON from **examples/animals.json** is being logged.

Explanation

Let's review what is happening.

Step 1. Fetch is called on a resource, **examples/animals.json**. Fetch returns a promise that resolves to a Response object. When the promise resolves, the response object is passed to `validateResponse`.

Step 2. `validateResponse` checks if the response is valid (is it a 200?). If it isn't, an error is thrown, skipping the rest of the `then` blocks and triggering the `catch` block. This is particularly important. Without this check bad responses are passed down the chain and could break later code that may rely on receiving a valid response. If the response is valid, it is passed to `readResponseAsJSON`.

Step 3. `readResponseAsJSON` reads the body of the response using the Response.json() method. This method returns a promise that resolves to JSON. Once this promise resolves, the JSON data is passed to `logResult`. (Can you think of what would happen if the promise from `response.json()` rejects?)

Step 4. Finally, the JSON data from the original request to **examples/animals.json** is logged by `logResult`.

For more information

- Response.json()
- Response methods
- Promise chaining

Solution code

To get a copy of the working code, navigate to the **02-fetching-a-resource** folder.

3. Fetch an image

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

To complete TODO 3a, replace the `showImage` function with the following code:

main.js

```
function showImage(responseAsBlob) {
  var container = document.getElementById('container');
  var imgElem = document.createElement('img');
  container.appendChild(imgElem);
  var imgUrl = URL.createObjectURL(responseAsBlob);
  imgElem.src = imgUrl;
}
```

To complete TODO 3b, finish writing the `readResponseAsBlob` function. The function should accept a response object as input. The function should return a promise that resolves to a Blob.

Note: This function will be very similar to `readResponseAsJSON`. Check out the `blob()` method documentation).
(You can check that you have done this correctly by navigating to **app/test/test.html**.)

To complete TODO 3c, replace the `fetchImage` function with the following code:

main.js

```
function fetchImage() {
  fetch('examples/kitten.jpg')
  .then(validateResponse)
  .then(readResponseAsBlob)
  .then(showImage)
  .catch(logError);
}
```

Save the script and refresh the page. Click **Fetch image.** You should see an adorable kitten on the page.

Explanation

In this example an image is being fetched, **examples/kitten.jpg**. Just like in the previous exercise, the response is validated with `validateResponse`. The response is then read as a Blob (instead of JSON as in section 2). An image element is created and appended to the page, and the image's `src` attribute is set to a data URL representing the Blob.

Note: The URL object's createObjectURL() method is used to generate a data URL representing the Blob. This is important to note. You cannot set an image's source directly to a Blob. The Blob must be converted into a data URL.

For more information

- Blobs
- Response.blob()
- URL object

Solution code

To get a copy of the working code, navigate to the **03-fetching-images** folder.

4. Fetch text

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

To complete TODO 4a, replace the `showText` function with the following code:

main.js

```
function showText(responseAsText) {
  var message = document.getElementById('message');
  message.textContent = responseAsText;
}
```

To complete TODO 4b, finish writing the `readResponseAsText` function.. This function should accept a response object as input. The function should return a promise that resolves to text.

Note: This function will be very similar to `readResponseAsJSON` and `readResponseAsBlob`. Check out the `text()` method documentation).
(You can check that you have done this correctly by navigating to **app/test/test.html**.)

To complete TODO 4c, replace the `fetchText` function with the following code:

```
function fetchText() {
  fetch('examples/words.txt')
  .then(validateResponse)
  .then(readResponseAsText)
  .then(showText)
  .catch(logError);
}
```

Save the script and refresh the page. Click **Fetch text**. You should see a message on the page.

Explanation

In this example a text file is being fetched, **examples/words.txt**. Like the previous two exercises, the response is validated with `validateResponse`. Then the response is read as text, and appended to the page.

Note: While it may be tempting to fetch HTML and append it using the `innerHTML` attribute, be careful. This can expose your site to cross-site scripting attacks!

For more information

- Response.text()

Solution code

To get a copy of the working code, navigate to the **04-fetching-text** folder.

Note: 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.

5. Using HEAD requests

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

5.1 Make a HEAD request

To complete TODO 5.1, replace the `headRequest` function with the following code:

main.js

```
function headRequest() {
  fetch('examples/words.txt', {
    method: 'HEAD'
  })
  .then(validateResponse)
  .then(readResponseAsText)
  .then(logResult)
  .catch(logError);
}
```

Save the script and refresh the page. Click **HEAD request**. What do you notice about the console log? Is it showing you the text in **examples/words.txt**, or is it empty?

Explanation

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

In this example we set the fetch request 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.

5.2 Optional: Find the size of a resource

Let's look at the Headers of the fetch response from section 5.1 to determine the size of **examples/words.txt**.

Complete the function called `logSize` in TODO 5.2. The function accepts a response object as input. The function should log the `content-length` of the response. To do this, you need to access the headers property of the response, and use the headers object's get method.

After logging the the `content-length` header, the function should then return the response.

Then replace the `headRequest` function with the following code:

```
function headRequest() {
  fetch('examples/words.txt', {
    method: 'HEAD'
  })
  .then(validateResponse)
  .then(logSize)
  .then(readResponseAsText)
  .then(logResult)
  .catch(logError);
}
```

Save the script and refresh the page. Click **HEAD request**. The console should log the size (in bytes) of **examples/words.txt** (it should be 74 bytes).

Explanation

In this example, the HEAD method is used to request the size (in bytes) of a resource (represented in the `content-length` header) without actually loading the resource itself. In practice this could be used to determine if the full resource should be requested (or even how to request it).

Optional: Find out the size of **examples/words.txt** using another method and confirm that it matches the value from the response header (you can look up how to do this for your specific operating system—bonus points for using the command line!).

For more information

- HTTP methods
- Fetch method signature
- Headers

Solution code

To get a copy of the working code, navigate to the **05-head-requests** folder.

6. Using POST requests

Fetch can also send data with POST requests.

6.1 Set up an echo server

For this example you need to run an echo server. From the **fetch-api-lab/app** directory run the following commands:

```
npm install
node echo-servers/echo-server-cors.js
```

You can check that you have successfully started the server by navigating to **app/test/test.html** and checking the 'echo server #1 running (with CORS)' task. If it is red, then the server is not running.

Explanation

In this step we install and run a simple server at **localhost:5000/** that echoes back the requests sent to it.

Note: If you need to, you can stop the server by pressing **Ctrl+C** from the command line.

6.2 Make a POST request

To complete TODO 6.2, replace the `postRequest` function with the following code:

main.js

```
function postRequest() {
  // TODO 6.3
  fetch('http://localhost:5000/', {
    method: 'POST',
    body: 'name=david&message=hello'
  })
  .then(validateResponse)
  .then(readResponseAsText)
  .then(logResult)
  .catch(logError);
}
```

Save the script and refresh the page. Click **POST request**. Do you see the sent request echoed in the console? Does it contain the name and message?

Explanation

To make a POST request with fetch, we use the `init` parameter to specify the method (similar to how we set the HEAD method in section 5). This is also where we set the **body** of the request. The body is the data we want to send.

Note: In production, remember to always encrypt any sensitive user data.
When data is sent as a POST request to **localhost:5000/**, the request is echoed back as the response. The response is then validated with `validateResponse`, read as text, and logged to the console.

In practice, this server would be a 3rd party API.

6.3 Use the FormData interface

You can use the FormData interface to easily grab data from forms.

In the `postRequest` function, replace TODO 6.3 with the following code:

main.js

```
var formData = new FormData(document.getElementById('myForm'));
```

Then replace the value of the `body` parameter with the `formData` variable.

Save the script and refresh the page. Fill out the form (the **Name** and **Message** fields) on the page, and then click **POST** request. Do you see the form content logged in the console?

Explanation

The `FormData` constructor can take in an HTML `form`, and create a `FormData` object. This object is populated with the form's keys and values.

For more information

- POST requests
- FormData

Solution code

To get a copy of the working code, navigate to the **06-post-requests** folder.

7. Optional: CORS and custom headers

7.1 Start a new echo server

Stop the previous echo server (by pressing **Ctrl+C** from the command line) and start a new echo server from the **fetch-lab-api/app** directory by running the following command:

```
node echo-servers/echo-server-no-cors.js
```

You can check that you have successfully started the server by navigating to **app/test/test.html** and checking the 'echo server #2 running (without CORS)' task. If it is red, then the server is not running.

Explanation

The application we run in this step sets up another simple echo server, this time at **localhost:5001/**. This server, however, is not configured to accept cross origin requests.

Note: You can stop the server by pressing **Ctrl+C** from the command line.

7.2 Fetch from the new server

Now that the new server is running at **localhost:5001/**, we can send a fetch request to it.

Update the `postRequest` function to fetch from **localhost:5001/** instead of **localhost:5000/**. Save the script, refresh the page, and then click **POST Request**.

You should get an error indicating that the cross-origin request is blocked due to the CORS `Access-Control-Allow-Origin` header being missing.

Update fetch in the `postRequest` function to use no-cors mode (as the error log suggests). Comment out the `validateResponse` and `readResponseAsText` steps in the fetch chain. Save the script and refresh the page. Then click **POST Request**.

You should get a response object logged in the console.

Explanation

Fetch (and XMLHttpRequest) follow the same-origin policy. This means that browsers restrict cross-origin HTTP requests from within scripts. A cross-origin request occurs when one domain (for example **http://foo.com/**) requests a resource from a separate domain (for example **http://bar.com/**).

Note: Cross-origin request restrictions are often a point of confusion. Many resources like images, stylesheets, and scripts are fetched across domains (i.e., cross-origin). However,

these are exceptions to the same-origin policy. Cross-origin requests are still restricted from *within scripts* .
Since our app's server has a different port number than the two echo servers, requests to either of the echo servers are considered cross-origin. The first echo server, however, running on **localhost:5000/**, is configured to support CORS. The new echo server, running on **localhost:5001/**, is not (which is why we get an error).

Using `mode: no-cors` allows fetching an opaque response. This prevents accessing the response with JavaScript (which is why we comment out `validateResponse` and `readResponseAsText`), but the response can still be consumed by other API's or cached by a service worker.

7.3 Modify request headers

Fetch also supports modifying request headers. Stop the **localhost:5001** (no CORS) echo server and restart the **localhost:5000** (CORS) echo server from section 6 (`node echo-servers/echo-server-cors.js`).

Update the `postRequest` function to fetch from **localhost:5000/** again. Remove the `no-cors` mode setting from the `init` object or update the mode to `cors` (these are equivalent, as `cors` is the default mode). Uncomment the `validateResponse` and `readResponseAsText` steps in the fetch chain.

Now use the Header interface to create a Headers object inside the `postRequest` function called `customHeaders` with the `Content-Type` header equal to `text/plain` . Then add a headers property to the `init` object and set the value to be the `customHeaders` variable. Save the script and refresh the page. Then click **POST Request**.

You should see that the echoed request now has a `Content-Type` of `plain/text` (as opposed to `multipart/form-data` as it had previously).

Now add a custom `Content-Length` header to the `customHeaders` object and give the request an arbitrary size. Save the script, refresh the page, and click **POST Request**. Observe that this header is not modified in the echoed request.

Explanation

The Header interface enables the creation and modification of Headers objects. Some headers, like `Content-Type` can be modified by fetch. Others, like `Content-Length` , are guarded and can't be modified (for security reasons).

7.4 Set custom request headers

Fetch supports setting custom headers.

Remove the `Content-Length` header from the `customHeaders` object in the `postRequest` function. Add the custom header `X-Custom` with an arbitrary value (for example `'X-CUSTOM': 'hello world'`). Save the script, refresh the page, and then click **POST Request**.

You should see that the echoed request has the `X-Custom` that you added.

Now add a `Y-Custom` header to the Headers object. Save the script, refresh the page, and click **POST Request**.

You should get an error similar to this in the console:

```
Fetch API cannot load http://localhost:5000/. Request header field y-custom is not all
owed by Access-Control-Allow-Headers in preflight response.
```

Explanation

Like cross-origin requests, custom headers must be supported by the server from which the resource is requested. In this example, our echo server is configured to accept the `X-Custom` header but not the `Y-Custom` header (you can open **echo-servers/echo-server-cors.js** and look for `Access-Control-Allow-Headers` to see for yourself). Anytime a custom header is set, the browser performs a preflight check. This means that the browser first sends an OPTIONS request to the server, to determine what HTTP methods and headers are allowed by the server. If the server is configured to accept the method and headers of the original request, then it is sent, otherwise an error is thrown.

For more information

- Cross Origin Resource Sharing (CORS)
- That's so fetch!

Solution code

To get a copy of the working code, navigate to the **solution** folder.

Congratulations!

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

Resources

- Fetch API Concepts
- Learn more about the Fetch API
- Learn more about Using Fetch
- Learn more about GlobalFetch.fetch()
- Get an Introduction to Fetch
- David Walsh's blog on fetch
- Jake Archibald's blog on fetch

Lab: Caching Files with Service Worker

Contents

Overview

1. Get set up

2. Cache the application shell

3. Serve files from the cache

4. Add network responses to the cache

5. Respond with custom 404 page

6. Respond with custom offline page

7. Delete outdated caches

Congratulations!

Concepts: Caching Files with Service Worker

Overview

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.

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
- How to remove unused caches on service worker activation

What you should know

- Basic JavaScript and HTML

- Familiarity with the concept and basic syntax of ES2015 Promises

What you will need

- Computer with terminal/shell access
- Connection to the internet
- A browser that supports `caches`
- A text editor

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open your browser and navigate to **localhost:8080/cache-api-lab/app**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **cache-api-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **images** folder contains sample images, each with several versions at different resolutions
- **pages** folder contains sample pages and a custom offline page
- **style** folder contains the app's cascading stylesheet
- **test** folder contains QUnit tests
- **index.html** is the main HTML page for our sample site/application
- **service-worker.js** is the service worker file where we set up the interactions with the cache

2. Cache the application shell

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

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

service-worker.js

```
var filesToCache = [
  '.',
  'style/main.css',
  'https://fonts.googleapis.com/css?family=Roboto:300,400,500,700',
  'images/still_life-1600_large_2x.jpg',
  'images/still_life-800_large_1x.jpg',
  'images/still_life_small.jpg',
  'images/still_life_medium.jpg',
  'index.html',
  'pages/offline.html',
  'pages/404.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);
    })
  );
});
```

Save the code and reload the page in the browser.Update the service worker and then open the cache storage in the browser. You should see the files appear in the table. You may need to refresh the page again for the changes to appear.

Open the first QUnit test page, **app/test/test1.html**, in another browser tab.

Note: Be sure to open the test page using the localhost address so that it opens from the server and not directly from the file system.
This page contains several tests for testing our app at each stage of the codelab. Passed tests are blue and failed tests are red. At this point, your app should pass the first two tests. These check that the cache exists and that it contains the app shell.

Caution: Close the test page when you're finished with it, otherwise you won't be able to activate the updated service worker in the next sections. See the Introduction to service worker text for an explanation.
Note: In Chrome, you can delete the cache in **DevTools**.

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,CSS, and image 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

- The Application Shell
- The install event - MDN

3. Serve 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 3 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 4 - Add fetched files to the cache

    }).catch(function(error) {

      // TODO 6 - Respond with custom offline page

    })
  );
});
```

Save the code and update the service worker in the browser (make sure you have closed the **test.html** page). Refresh the page to see the network requests being logged to the console. Now take the app offline 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 (with `caches.match`) 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

4. Add network responses to the cache

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

Replace TODO 4 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) {

  // TODO 5 - Respond with custom 404 page

  return caches.open(staticCacheName).then(function(cache) {
    if (event.request.url.indexOf('test') < 0) {
      cache.put(event.request.url, response.clone());
    }
    return response;
  });
});
```

Save the code. Take the app back online and update the service worker. Visit at least one of the links on the homepage, then take the app offline again. Now if you revisit the pages they should load normally! Try navigating to some pages you haven't visited before.

Take the app back online and open **app/test/test1.html** in a new tab. Your app should now pass the third test that checks whether network responses are being added to the cache. Remember to close the test page when you're done.

Explanation

Here we are taking the responses returned from the network requests and putting them into the cache.

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 article for an explanation.

We have wrapped the code to cache the response in an `if` statement to ensure we are not caching our test page.

For more information

- Cache.put - MDN

5. Respond with custom 404 page

Below TODO 5 in **service-worker.js**, write the code to respond with the **404.html** page from the cache if the response status is `404`. You can check the response status with `response.status`.

To test your code, save what you've written and then update the service worker in the browser. Click the **Non-existent file** link to request a resource that doesn't exist.

Explanation

Network response errors do not throw an error in the `fetch` promise. Instead, `fetch` returns the response object containing the error code of the network error. This means we handle network errors in a `.then` instead of a `.catch`. However, if the `fetch` cannot reach the network (user is offline) an error is thrown in the promise and the `.catch` executes.

Note: When intercepting a network request and serving a custom response, the service worker does not redirect the user to the address of the new response. The response is served at the address of the original request. For example, if the user requests a nonexistent file at **www.example.com/non-existent.html** and the service worker responds with a custom 404 page, **404.html**, the custom page will display at **www.example.com/non-existent.html**, not **www.example.com/404.html**.

For more information

- Response.status - MDN
- Response status codes - MDN
- Response.ok - MDN

Solution code

The solution code can be found in the **05-404-page** directory.

6. Respond with custom offline page

Below TODO 6 in the `.catch` in **service-worker.js**, write the code to respond with the **offline.html** page from the cache. The catch will trigger if the fetch to the network fails.

To test your code, save what you've written and then update the service worker in the browser. Take the app offline and navigate to a page you haven't visited before to see the custom offline page.

Explanation

If `fetch` cannot reach the network, it throws an error and sends it to a `.catch`.

Solution code

The solution code can be found in the **06-offline-page** directory.

7. Delete outdated caches

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

Replace TODO 7 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';
```

Save the code and update the service worker in the browser. Inspect the cache storage in your browser. You should see just the new cache. The old cache, `pages-cache-v1`, has been removed.

Open **app/test/test2.html** in a new browser tab. The test checks whether `pages-cache-v1` has been deleted and that `pages-cache-v2` has been created.

Explanation

We delete old caches in the `activate` event to make sure that we aren't deleting caches before the new 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

The solution code can be found in the **solution** directory.

Congratulations!

You have learned how to use the Cache API in the service worker.

What we've covered

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.

Resources

Learn more about caching and the Cache API

- Cache - MDN
- The Offline Cookbook

Learn more about using service workers

- Using Service Workers - MDN

Lab: IndexedDB

Contents

Overview
1. Get set up
2. Check for support
3. Creating the database and adding items
4. Searching the database
5. Optional: Processing orders
Congratulations!

Concepts: Working with IndexedDB

Overview

This lab guides you through the basics of the IndexedDB API using Jake Archibald's IndexedDB Promised library. The IndexedDB Promised library 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.

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 (or CRUD)
- How to use cursors
- (Optional) How to use the `getAll()` method

What you should know

- Basic JavaScript and HTML
- JavaScript Promises

What you will need

- Computer with terminal/shell access
- Chrome (the unit tests have a Chrome dependency)

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open your browser and navigate to **localhost:8080/indexed-db-lab/app**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **indexed-db-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **js/main.js** is where we will write the scripts to interact with the database
- **js/idb.js** is the IndexedDB Promised library
- **test/test.html** is a QUnit test page
- **index.html** is the main HTML page for our sample site/application, and which contains some forms for interacting with our IndexedDB database

2. Check for support

Because IndexedDB isn't supported by all browsers, we need to check that the user's browser supports it before using it.

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

main.js

```
if (!('indexedDB' in window)) {
  console.log('This browser doesn\'t support IndexedDB');
  return;
}
```

3. Creating the database and adding items

3.1 Open a database

Create the database for your app.

In **js/main.js**, replace `var dbPromise;` with the following code:

main.js

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

Save the file and refresh the page in the browser. Open IndexedDB in the developer tools and confirm that your database exists.

The IndexedDB UI in DevTools doesn't always accurately reflect what's in the database. In Chrome, if you don't see your changes, try right-clicking on **IndexedDB** in the **Application** tab and then click **Refresh IndexedDB**. If it still doesn't update, then try closing and re-opening DevTools.

Explanation

`idb.open` takes a database name, version number, and optional callback function for performing database updates (not included in the above code). The version number determines whether the upgrade callback function is called. If the version number is greater than the version number of the database existing in the browser, then the upgrade callback is executed.

Note: If at any point in the codelab your database gets into a bad state, you can delete it in Chrome DevTools by going to the **Application** tab, clicking on the database name under IndexedDB, and clicking the **Delete database** button. Alternatively, you can click **Clear storage** (in the Application tab) and then click the **Clear site data** button. In all browsers you can also delete the database from the console with the following command: `indexedDB.deleteDatabase('couches-n-things');` .

3.2 Create an object store

Let's create an object store in the database to hold the furniture objects.

Replace `var dbPromise = idb.open('couches-n-things', 1);` in **main.js** with the following:

main.js

```
var dbPromise = idb.open('couches-n-things', 2, function(upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
      // a placeholder case so that the switch block will
      // execute when the database is first created
      // (oldVersion is 0)
    case 1:
      console.log('Creating the products object store');
      upgradeDb.createObjectStore('products', {keyPath: 'id'});

    // TODO 4.1 - create 'name' index

    // TODO 4.2 - create 'price' and 'description' indexes

    // TODO 5.1 - create an 'orders' object store

  }
});
```

Save the code and reload the page in the browser. Open IndexedDB in your browser's developer tools and expand the `couches-n-things` database. You should see the empty `products` object store.

Open the QUnit test page, **localhost:8080/app/test/test.html**, in another browser tab. This page contains several tests for testing our app at each stage of the codelab. Passed tests are blue and failed tests are red. Your app should pass the first test that checks whether the `products` object store exists in the database. Note that you may not be able to delete the database while the testing page is open.

Explanation

To ensure database integrity, object stores and indexes can only be created during database upgrades. This means they are created inside the upgrade callback function in `idb.open`, which executes only if the version number (in this case it's `2`) is greater than the existing version in the browser, or if the database doesn't exist. The callback is passed the `UpgradeDB` object (see the documentation for details), 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. The database already exists for us, but we need a `case 0` in case we delete the database, or in case someone else uses our app on their own machine.

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.

Note: 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

- `idb` - Github
- `createObjectStore` method

3.3 Add objects to the object store

Add some sample furniture items to the object store.

Replace TODO 3.3 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
    }
  ];
  return Promise.all(items.map(function(item) {
      console.log('Adding item: ', item);
      return store.add(item);
    })
  ).catch(function(e) {
    tx.abort();
    console.log(e);
  }).then(function() {
    console.log('All items added successfully!');
  });
});
```

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 under `couches-n-things` in the developer tools. Remember you may need to refresh IndexedDB to see the changes by right clicking **IndexedDB** and clicking **Refresh IndexedDB**. If that doesn't work, then try collapsing IndexedDB and closing and re-opening DevTools.

Reload the test page. The app should now pass the next test that checks whether the objects have been added to the `products` object store.

Explanation

All database operations must be carried out within a transaction. In the code we just wrote, we first open the transaction on the database object and then open the object store on the transaction. Now when we call `store.add` on that object store, the operation happens inside the transaction.

We add each object to the store inside a `Promise.all`. This way if any of the `add` operations fail, we can catch the error and abort the transaction. Aborting the transaction rolls back all the changes that happened in the transaction so that if any of the events fail to add, none of them will be added to the object store. This ensures the database is not left in a partially updated state.

Note: Specify the transaction mode as `readwrite` when making changes to the database (that is, for changes that use the `add`, `put`, or `delete` methods).

For more information

- Transactions - MDN
- Add method - MDN

4. Searching the database

4.1 Create indexes on the object store

Create some indexes on your object store.

Close the test page. The database version can't be changed while another page is using the database.

Replace TODO 4.1 in **main.js** with the following code:

main.js

```
case 2:
  console.log('Creating a name index');
  var store = upgradeDb.transaction.objectStore('products');
  store.createIndex('name', 'name', {unique: true});
```

Change the version number to 3 in the call to `idb.open`. The full `idb.open` method should look like this:

main.js

```
var dbPromise = idb.open('couches-n-things', 3, function(upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
      // a placeholder case so that the switch block will
      // execute when the database is first created
      // (oldVersion is 0)
    case 1:
      console.log('Creating the products object store');
      upgradeDb.createObjectStore('products', {keyPath: 'id'});
    case 2:
      console.log('Creating a name index');
      var store = upgradeDb.transaction.objectStore('products');
      store.createIndex('name', 'name', {unique: true});

    // TODO 4.2 - create 'price' and 'description' indexes

    // TODO 5.1 - create an 'orders' object store

  }
});
```

Note: We did not include break statements in the switch block so that all of the latest updates to the database will execute even if the user is one or more versions behind.
Save the file and reload the page in the browser. Confirm that the `name` index displays in the `products` object store in the developer tools. You may need to refresh IndexedDB to see your changes.

Open the test page. The app should now pass the next test that checks whether the `name` index exists.

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

4.2 Create `price` and `description` indexes

This step is for you to complete on your own. In **main.js**, write a `case 3` in `idb.open` to add `price` and `description` indexes to the `products` object store. Do not include the optional `{unique: true}` argument since these values do not need to be unique. The code should be very similar to the code in the previous step. Remember to change the version number of the database to 4 before testing the code.

Before testing your code, close the unit test page in the browser. Save your changes to the code and refresh the page in the browser. Confirm that the `price` and `description` indexes display in the `products` object store in the developer tools. You may need to clear the database for the changes to appear in DevTools. Otherwise, you can just re-open the unit test page. If your app is passing the next two tests which check if the `price` and `description` indexes exist, you've done this step correctly.

4.3 Use the get method

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

Add the following code to the `getByName` function in **main.js**:

main.js

```
return dbPromise.then(function(db) {
  var tx = db.transaction('products', 'readonly');
  var store = tx.objectStore('products');
  var index = store.index('name');
  return index.get(key);
});
```

Save the code and refresh the page in the browser.

Note: Make sure the items we added to the database in the previous step are still in the database. If the database is empty, click **Add Products** to populate it. Don't worry about adding things twice. IndexedDB will throw errors in the console if you try to add items that already exist and won't add them to the store.

Enter an item name from step 3.3 (try "Chair") into the **By Name** field and click **Search** next to the text box. The corresponding furniture item should display on the page.

Refresh the test page. The app should pass the next test, which checks if the `getByName` function returns a database object.

Note: The `get` method (and consequently `getByName`) is case sensitive.

Explanation

The **Search** button calls the `displayByName` function, which passes the user input string to the `getByName` function. The code we just added calls the `get` method on the `name` index to retrieve an item by its `name` property.

For more information

- `get` method - MDN

4.4 Use a cursor object

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

Replace TODO 4.4a in **main.js** with the following code:

main.js

```
var lower = document.getElementById('priceLower').value;
var upper = document.getElementById('priceUpper').value;
var lowerNum = Number(document.getElementById('priceLower').value);
var upperNum = Number(document.getElementById('priceUpper').value);

if (lower === '' && upper === '') {return;}
var range;
if (lower !== '' && upper !== '') {
  range = IDBKeyRange.bound(lowerNum, upperNum);
} else if (lower === '') {
  range = IDBKeyRange.upperBound(upperNum);
} else {
  range = IDBKeyRange.lowerBound(lowerNum);
}
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;
});
```

Save the code and refresh the page in the browser. Enter some prices into the 'price' text boxes (without a currency symbol; try 200 and 500) and click **Search**. Items should appear on the page ordered by price.

Optional: On your own time, replace TODO 4.4b 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. To test your code, try putting "A light, high-stool" into the **By Description** input and clicking **Search**.

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

The solution code for the lab up to this point can be found in the **04-4-get-data** directory.

5. Optional: Processing orders

In this section we create an `orders` object store to contain a user's orders. We take a sample order and check if the quantity of each item in the `products` object store is enough to fulfill the order. If we have enough in stock, we process the orders, subtracting the amount ordered from the quantity of each corresponding item in the `products` object store.

5.1 Create an orders object store

This step is for you to complete on your own. Create an object store to hold pending orders.

To complete TODO 5.1 in **main.js**, write a case 4 that adds an `orders` object store to the database. Make the `keyPath` the `id` property. This is very similar to creating the `products` object store in `case 1`. Remember to change the version number of the database to 5 so the callback executes.

Before testing your code, close the unit test page. Save the code and refresh the page in the browser. Confirm that the object store displays in the developer tools.

Open the test page. Your app should pass the next test which tests if the `orders` object store exists.

5.2 Add sample orders

This step is for you to complete on your own. In the `addOrders` function in **main.js**, write the code to add the following items to the `orders` object store:

Hint: This code will be very similar to the `addProducts` function that we wrote at the start of the lab.

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
  }
];
```

Save the code and refresh the page in the browser. Click **Add Orders** and refresh the page again. Confirm that the objects show up in the `orders` store in the developer tools.

Refresh the test page. Your app should now pass the next test which checks if the sample orders were added to the `orders` object store.

5.3 Display orders

This step is for you to complete on your own. To complete TODO 5.3 in **main.js**, write 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 and you should display the name instead of the price. The code to insert the `s` variable into the HTML is already written.

Save the code and refresh the page in the browser. Click **Show Orders** to display the orders on the page.

5.4 Get all orders

This step is for you to complete on your own. To complete TODO 5.4 in the `getOrders` function in **main.js**, write 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 in the `fulfillOrders` function.

Hint: Return the call to `dbPromise` otherwise the orders array will not be passed to the `processOrders` function.
Refresh the test page. Your app should now pass the next test, which checks if the `getOrders` function gets objects from the `orders` object store.

5.5 Process the orders

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

Replace TODO 5.5 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

5.6 Decrement quantity

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

Replace TODO 5.6 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>';
    throw 'Out of stock!';
  }
  item.quantity = qtyRemaining;
  resolve(item);
});
```

Refresh the test page. Your app should now pass the next test, which checks if the `decrementQuantity` function subtracts the quantity ordered from the quantity available.

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 we update the quantity and return the object.

For more information

- `new Promise` - MDN

5.7 Update the `products` object store

Finally, we must update the `products` object store with the new quantities of each item. This step is for you to complete on your own.

Replace TODO 5.7 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 to update each item in the store. A few hints:

- Remember to make the transaction mode `'readwrite'`
- Remember to return `tx.complete` after putting the items into the store

Save the code and refresh the page in the browser. Check the quantity property of the cabinet, armchair, and couch items in the products object store. Click **Fulfill** in the page, refresh, and check the quantities again. They should be reduced by the amount of each product that was ordered.

Refresh the test page. Your app should now pass the last test, which checks whether the `updateProductsStore` function updates the items in the `products` object store with their reduced quantities.

Solution code

The solution code can be found in the **solution** directory.

Congratulations!

You have learned the basics of working with IndexedDB.

What we've covered

- How to create, read, update and delete data in the database
- The `getAll` method
- How to use cursors to iterate over the data

Lab: Auditing with Lighthouse

Contents

Overview
1. Get set up
2. Install Lighthouse
3. Test the app
4. Adding a manifest file
5. Adding a service worker
6. Test the updated app
7. Optional: Run Lighthouse from the command line
Congratulations!

Concepts: Lighthouse PWA Analysis Tool

Overview

This lab shows you how you can use Lighthouse, an open-source tool from Google, to audit a web app for PWA features. Lighthouse provides a set of metrics to help guide you in building a PWA with a full application-like experience for your users.

What you will learn

- How to use Lighthouse to audit your progressive web apps

What you should know

- Basic JavaScript and HTML

What you need before you begin

- Connection to the internet
- Chrome 52+ browser
- Node v6+
- A text editor

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open your browser and navigate to **localhost:8080/lighthouse-lab/app**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **lighthouse-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **css/main.css** is the cascading stylesheet for the application
- **images** folder contains images for the app and home screen icon
- **index.html** is the main HTML page for our sample site/application

2. Install Lighthouse

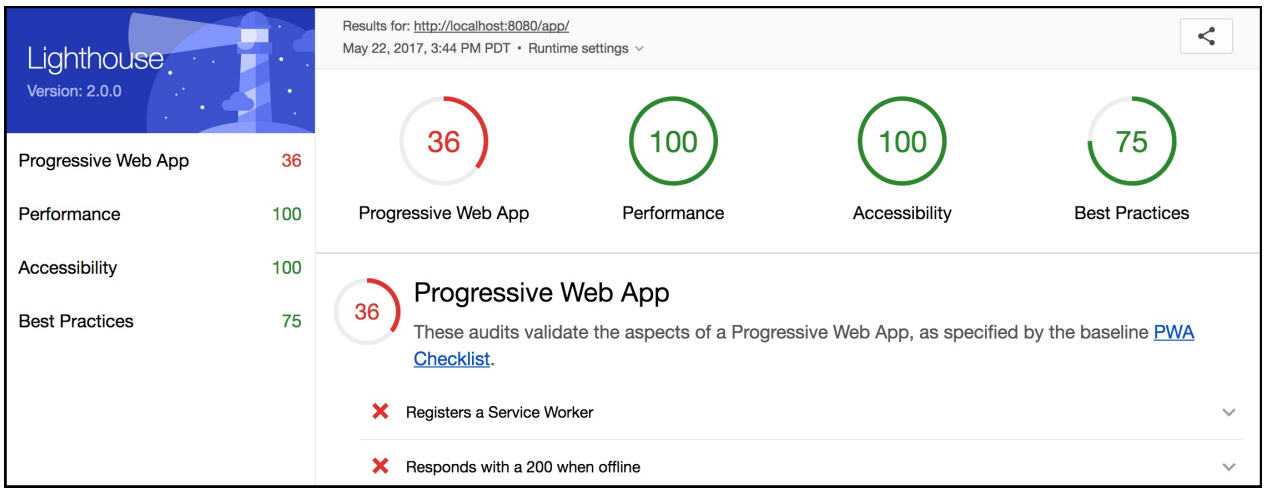
Lighthouse is available as a Chrome extension for Chrome 52 and later.

Download the Lighthouse Chrome extension from the Chrome Web Store. When installed it places an  icon in your taskbar.

3. Test the app

In the browser (on the **app** page) click the Lighthouse icon and choose **Generate report** (you may be prompted to close Developer Tools if they are open).

Lighthouse runs the report and generates an HTML page with the results. The report page should look similar to this:



Note: The UI for Lighthouse is still being updated, so your report may not look exactly like this one.
Looks like we have a pretty low score (your score may not match exactly). Take a moment to look through the report and see what is missing.

4. Adding a manifest file

The report indicates that we need a manifest file.

4.1 Create the manifest file

Create an empty file called **manifest.json** in the **app** directory.

Replace TODO 4.1 in **index.html** with the following:

index.html

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

4.2 Add manifest code

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

manifest.json

```
{
  "name": "Demo Blog Application",
  "short_name": "Blog",
  "start_url": "index.html",
  "icons": [{
        "src": "images/touch/icon-128x128.png",
        "sizes": "128x128",
        "type": "image/png"
      }, {
        "src": "images/touch/icon-192x192.png",
        "sizes": "192x192",
        "type": "image/png"
      }, {
        "src": "images/touch/icon-256x256.png",
        "sizes": "256x256",
        "type": "image/png"
      }, {
        "src": "images/touch/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"
}
```

4.3 Add tags for other browsers

Replace TODO 4.3 in **index.html** with the following:

index.html

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

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

<!-- Tile for Win8 -->
<meta name="msapplication-TileColor" content="#3372DF">
<meta name="msapplication-TileImage" content="images/touch/icon-192x192.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 lets us satisfy the manifest related requirements of Lighthouse (and a PWA).

For more information

- Add to home screen
- Web app manifests

5. Adding a service worker

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

5.1 Register a service worker

Create an empty JavaScript file in the root directory (**app**) and name it **service-worker.js**. This is going to be our service worker file.

Now replace TODO 5.1 in **index.html** with the following and save the file:

index.html

```
<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>
```

5.2 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 (which should be at **app/service-worker.js**):

service-worker.js

```
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('static-cache-v1')
    .then(function(cache) {
      return cache.addAll([
        '.',
        'index.html',
        'css/main.css',
        'https://fonts.googleapis.com/css?family=Roboto:300,400,500,700',
        'images/still_life-1600_large_2x.jpg',
        'images/still_life-800_large_1x.jpg',
        'images/still_life_medium.jpg',
        'images/still_life_small.jpg'
      ]);
    })
  );
});

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

Save the file and refresh the page (for the app, not the Lighthouse page). Check the console and confirm that the service worker has registered successfully.

Explanation

We have created a service worker for our app and registered it. Here is what it does:

1. The first block (`install` event listener) caches the files our app's files, so that they are saved locally. This lets us 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 without needing to make a network request. This lets us respond with a 200 even when offline.

Once we have loaded the app initially, all the files needed to run the app 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.

Solution code

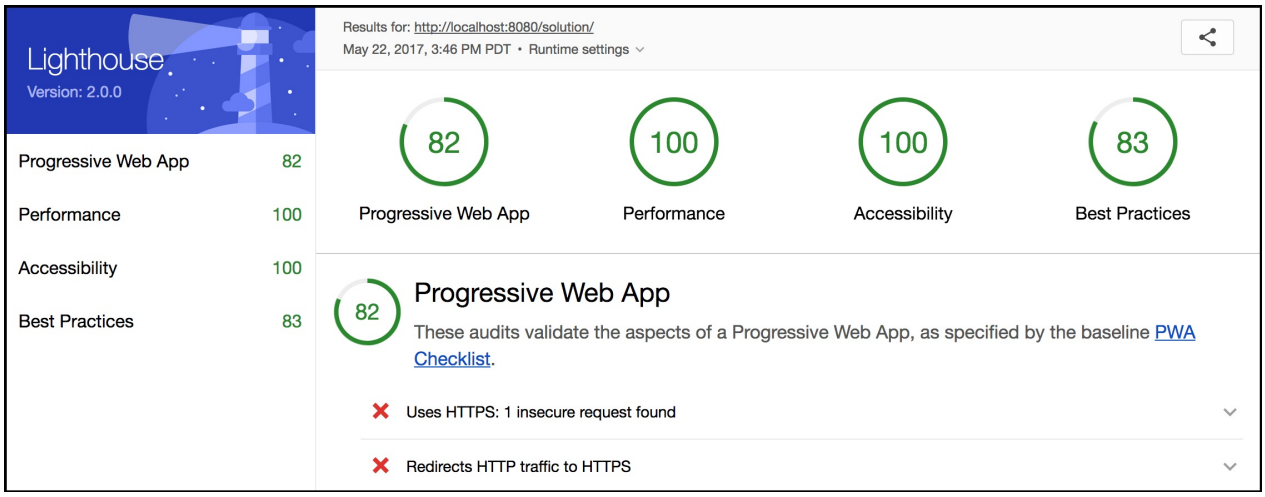
To get a copy of the working code, navigate to the **solution** folder.

6. Test 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 generate a report (you may be prompted to close Developer Tools if they are still open).

Now we should have passed many more tests.

Note: You may need to disable the browser cache to see the improved results. Then refresh the app and run Lighthouse again.
The report should look something like this:



Now our score is much better (your score may not match exactly).

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.

7. Optional: Run Lighthouse from the command line

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

If you haven't already, download Node and select the Long Term Support (LTS) version that best suits your environment and operating system (Lighthouse requires Node v6 or greater).

Install Lighthouse's Node module from the command line:

```
npm install -g lighthouse
```

Run Lighthouse on a demo progressive web app:

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

Or on the app that you just made (note that your localhost port may be different):

```
lighthouse http://localhost:8080/lighthouse-lab/app/
```

You can check Lighthouse flags and options with:

```
lighthouse --help
```

The lighthouse command line tool will generate an HTML (the same as the Chrome extension) in the working directory. You can then open the file with your browser.

Congratulations!

You have learned how to use the Lighthouse tool to audit your progressive web apps.

Lab: Gulp Setup

Contents

Overview

1. Get set up

2. Install global tools

3. Prepare the project

4. Minify JavaScript

5. Prefix CSS

6. Automate development tasks

7. Congratulations!

Concepts: Introduction to Gulp

Overview

This lab shows you how you can automate tasks with gulp, a build tool and task runner.

What you will learn

- How to set up gulp
- How to create tasks using gulp plugins
- Ways to automate your development

What you should know

- Basic JavaScript, HTML, and CSS
- Some experience using a command line interface

What you will need

- Computer with terminal/shell access
- Connection to the internet

- A text editor
- Node and npm

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **gulp-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **js/main.js** and **styles/main.css** are sample resources that we use to experiment
- **index.html** is the main HTML page for the sample site/application
- **gulpfile.js** is the file that gulp uses to execute tasks, and where you will write your code

2. Install global tools

Gulp is available as a Node package. In this section we install the gulp command line tool on your system.

To install the gulp command line tool, run the following in the command line:

```
npm install --global gulp-cli
```

Explanation

This command installs the gulp command line tool (globally) using npm. We use the command line tool to actually execute gulp.

3. Prepare the project

All projects that use gulp need to have the gulp package installed locally.

From **app/** (the project root), run the following in the command line:

```
npm init -y
```

Note that a **package.json** file was created. Open the file and inspect it.

From the same directory, run the following in the command line:

```
npm install gulp --save-dev
```

Note that a **node_modules** directory has been added to the project with various packages. Also note that **package.json** now lists "gulp" as a dependency.

Note: Some text editors hide files and directories that are listed in the **.gitignore** file. Both **node_modules** and **build** are in our **.gitignore**. If you have trouble viewing these during the lab, just delete the **.gitignore** file.
In **gulpfile.js**, replace the TODO 3 comment with the following:

gulpfile.js

```
var gulp = require('gulp');
```

Explanation

We start by generating **package.json** with `npm init` (the `-y` flag uses default configuration values for simplicity). This file is used to keep track of the packages that your project uses, including gulp and its dependencies.

The next command installs the gulp package and its dependencies in the project. These are put in a **node_modules** folder. The `--save-dev` flag adds the corresponding package (in this case gulp) to **package.json**. Tracking packages like this allows quick re-installation of all the packages and their dependencies on future builds (the `npm install` command will read **package.json** and automatically install everything listed).

Finally we add code to **gulpfile.js** to include the gulp package. The **gulpfile.js** file is where all of the gulp code should go.

4. Minify JavaScript

This exercise implements a simple task to minify (also called "uglify" for JavaScript) the **app/js/main.js** JavaScript file.

From **app/**, run the following in the command line:

```
npm install gulp-uglify --save-dev
```

Now replace TODO 4.1 in **gulpfile.js** with the following code:

gulpfile.js

```
var uglify = require('gulp-uglify');
```

Replace TODO 4.2 with the following code:

gulpfile.js

```
gulp.task('minify', function() {
  gulp.src('js/main.js')
  .pipe(uglify())
  .pipe(gulp.dest('build'));
});
```

Save the file. From **app/**, run the following in the command line:

```
gulp minify
```

Open **app/js/main.js** and **app/build/main.js**. Note that the JavaScript from **app/js/main.js** has been minified into **app/build/main.js**.

Explanation

We start by installing the **gulp-uglify** package (this also updates the **package.json** dependencies). This enables minification functionality in our gulp process.

Then we include this package in the **gulpfile.js** file, and add code to create a `minify` task. This task gets the **app/js/main.js** file, and pipes it to the `uglify` function (which is defined in the **gulp-uglify** package). The `uglify` function minifies the file, pipes it to the `gulp.dest` function, and creates a **build** folder containing the minified JavaScript.

5. Prefix CSS

In this exercise, you add vendor prefixes to the **main.css** file.

Read the documentation for gulp-autoprefixer. Using section 4 of this lab as an example, complete the following tasks:

1. Install the gulp-autoprefixer package
2. Require the package in **gulpfile.js**
3. Write a task in **gulpfile.js** called `processCSS`, that adds vendor prefixes to the **app/styles/main.css** and puts the new file in **app/build/main.css**

Test this task by running the following (from **app/**) in the command line:

```
gulp processCSS
```

Open **app/styles/main.css** and **app/build/main.css**. Does the `box-container` class have vendor prefixes for the `display: flex` property?

Optional: Read the gulp-sourcemaps documentation and incorporate sourcemap generation in the `processCSS` task (not in a new task).

Hint: The gulp-autoprefixer documentation has a useful example. Test by rerunning the `processCSS` task, and noting the sourcemap comment in the **app/build/main.css** file.

6. Automate development tasks

6.1 Define default tasks

Usually we want to run multiple tasks each time we rebuild an application. Rather than running each task individually, they can be set as default tasks.

Replace TODO 6.1 in **gulpfile.js** with the following:

gulpfile.js

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

Now delete the **app/build** folder and run the following in the command line (from **app/**):

```
gulp
```

Note that both the `minify` and `processCSS` tasks were run with that single command (check that the **app/build** directory has been created and that **app/build/main.js** and **app/build/main.css** are there).

Explanation

Default tasks are run anytime the `gulp` command is executed.

6.2 Set up gulp watch

Even with default tasks, it can become tedious to run tasks each time a file is updated during development. gulp.watch watches files and automatically runs tasks when the corresponding files change.

Replace TODO 6.2 in **gulpfile.js** with the following:

gulpfile.js

```
gulp.task('watch', function() {
  gulp.watch('styles/*.css', ['processCSS']);
});
```

Save the file. From **app/**, run the following in the command line:

```
gulp watch
```

Add a comment to **app/styles/main.css** and save the file. Open **app/build/main.css** and note the real-time changes in the corresponding build file.

TODO: Now update the `watch` task in **gulpfile.js** to watch **app/js/main.js** and run the `minify` task anytime the file changes. Test by editing the value of the variable `future` in **app/js/main.js** and noting the real-time change in **app/build/main.js**. Don't forget to save the file and rerun the `watch` task.

Note: The watch task continues to execute once initiated. You need to restart the task in the command line whenever you make changes to the task. If there is an error in a file being watched, the watch task terminates, and must be restarted. To stop the task, use **Ctrl+c** in the command line or close the command line window.

Explanation

We created a task called `watch` that watches all CSS files in the **styles** directory, and all the JS files in the **js** directory. Any time any of these files changes (and is saved), the corresponding task (`processCSS` or `minify)` executes.

6.3 Set up BrowserSync

You can also automate the setup of a local testing server.

From **app/**, run the following in the command line:

```
npm install browser-sync --save-dev
```

Replace TODO 6.3a in **gulpfile.js** with the following:

gulpfile.js

```
var browserSync = require('browser-sync');
```

Now replace TODO 6.3b in **gulpfile.js** with the following:

gulpfile.js

```
gulp.task('serve', function() {
  browserSync.init({
    server: '.',
    port: 3000
  });
});
```

Save the file. Now run the following in the command line (from **app/**):

```
gulp serve
```

Your browser should open **app/** at **localhost:3000** (if it doesn't, open the browser and navigate there).

Explanation

The gulp browsersync package starts a local server at the specified directory. In this case we are specifying the target directory as '**.**', which is the current working directory (**app/**). We also specify the port as 3000.

6.4 Put it all together

Let's combine everything learned so far.

TODO: Change the default tasks from `minify` and `processCSS` to `serve`.

TODO: Update the `serve` task to the following code:

```
gulp.task('serve', ['processCSS'], function() {
  browserSync.init({
    server: '.',
    port: 3000
  });
  gulp.watch('styles/*.css', ['processCSS']).on('change', browserSync.reload);
  gulp.watch('*.html').on('change', browserSync.reload);
});
```

Close the app from the browser and delete **app/build/main.css**. From **app/**, run the following in the command line:

```
gulp
```

Your browser should open **app/** at **localhost:3000** (if it doesn't, open the browser and navigate there). Check that the **app/build/main.css** has been created. Change the color of the blocks in **app/styles/main.css** and check that the blocks change color in the page.

Explanation

In this example we changed the default task to `serve` so that it runs when we execute the `gulp` command. The `serve` task has `processCSS` as a *dependent task* . This means that the `serve` task will execute the `processCSS` task before executing itself. Additionally, this task sets a watch on CSS and HTML files. When CSS files are updated, the `processCSS` task is run again and the server reloads. Likewise, when HTML files are updated (like **index.html**), the browser page reloads automatically.

Optional: In the `serve` task, add `minify` as a dependent task. Also in `serve` , add a watcher for **app/js/main.js** that executes the `minify` task and reloads the page whenever the **app/js/main.js** file changes. Test by deleting **app/build/main.js** and re-executing the `gulp` command. Now **app/js/main.js** should be minified into **app/build/main.js** and it should update in real time. Confirm this by changing the console log message in **app/js/main.js** and saving the file - the console should log your new message in the app.

Congratulations!

You have learned how to set up gulp, create tasks using plugins, and automate your development!

Resources

- Gulp's Getting Started guide
- List of gulp Recipes
- Gulp Plugin Registry

Lab: Workbox

Contents

Overview

1. Get set up

2. Write a basic service worker using workbox-sw

3. Inject a manifest into the service worker

4. Register and test the service worker

5. Add routes to the service worker

6. Use a customized networkFirst cache strategy

7. Optional: Add a customized cacheFirst cache strategy

Congratulations!

Overview

Workbox is the successor to `sw-precache` and `sw-toolbox`. It is a collection of libraries and tools used for generating a service worker, precaching, routing, and runtime-caching. Workbox also includes modules for easily integrating background sync and Google Analytics into your service worker.

In this lab, you'll use the **workbox-sw.js** library and the `workbox-build` Node.js module to build an offline-capable progressive web app (PWA) .

What you'll learn

- How to write a service worker using the **workbox-sw.js** library
- How to add routes to your service worker using **workbox-sw.js**
- How to use the predefined caching strategies provided in **workbox-sw.js**
- How to augment the **workbox-sw.js** caching strategies with custom logic
- How to generate a production-grade service worker with `workbox-build`

What you should already know

- Basic HTML, CSS, and JavaScript
- ES2015 Promises
- How to run commands from the command line
- Some familiarity with service workers is recommended
- Familiarity with Node.js and gulp is recommended

What you will need

- Computer with terminal/shell access
- Connection to the internet
- A browser that supports service worker
- A text editor
- Node.js and npm

1. Get set up

If you have not downloaded the repository and installed the LTS version of Node.js, follow the instructions in Setting up the labs (you don't need to set up an HTTP server for this lab).

1.1 Install project dependencies and start the server

Navigate to the **project** directory via the command line:

```
cd workbox-lab/project/
```

Run the following command to install the project dependencies:

```
npm install
npm install gulp --global
```

Then build and serve the app with the gulp `serve` task:

```
gulp serve
```

You can terminate the `gulp serve` process at any time with `Ctrl-c`.

Explanation

The `npm install` command installs the project dependencies based on the configuration in **package.json**. Open **project/package.json** and examine its contents.

The important package is `workbox-build`, which is a module used to generate a list of assets that should be precached in a service worker. You'll see how that's used in a later step.

The remaining dependencies in **package.json** are used by gulp in **gulpfile.js**.

Open **gulpfile.js** and examine its contents. The `gulp serve` command runs the `serve` task. The `serve` task "builds" the app; in this case by copying the **src** files into a **build** directory. The **build** directory is then served and `watch`ed. The `watch` functionality ensures that the app is rebuilt whenever **src** files are updated.

1.3 Open the app and explore the code

Open the app in your browser by navigating to **localhost:8081/**. The app is a news site containing some "trending articles" and "archived posts". We will be performing different runtime caching strategies based on whether the request is for a trending article or archived post.

Open the **workbox-lab/project** folder in your text editor. The **project** folder is where you will be building the lab.

This folder contains:

- **src/images** folder contains sample images
- **src/js/animation.js** is a javascript file for page animations
- **src/pages** folder contains sample pages
- **src/style/main.css** is the stylesheet for the app's pages
- **src/sw.js** is where we will write our source service worker using **workbox-sw.js**
- **src/index.html** is the home page HTML file
- **gulpfile.js** configures gulp tasks
- **package.json** and **package-lock.json** track Node.js dependencies

All of the **src** files were also copied over to the **build** folder as mentioned in the previous section.

2. Write a basic service worker using workbox-sw

Now that we have the starting app working, let's start writing the service worker.

In the empty source service worker file, **src/sw.js**, add the following snippet:

src/sw.js

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/3.0.0/workbox-sw.js
');

if (workbox) {
  console.log(<code>Yay! Workbox is loaded </code>);

  workbox.precaching.precacheAndRoute([]);

} else {
  console.log(<code>Boo! Workbox didn't load </code>);
}
```

Save the file. Observe that our production service worker (**build/sw.js**) was automatically updated by gulp `watch`.

Explanation

In this code, the `importScripts` call imports the **workbox-sw.js** library from a Content Delivery Network (CDN). Once the library is loaded, the `workbox` object gives our service worker access to all the Workbox modules.

The `precacheAndRoute` method of the `precaching` module takes a precache "manifest" (a list of file URLs with "revision hashes") to cache on service worker installation. It also sets up a cache-first strategy for the specified resources, serving them from the cache by default.

Currently, the array is empty, so no files will be cached.

Rather than adding files to the list manually, `workbox-build` can generate the manifest for us. Using a tool like `workbox-build` has multiple advantages:

1. The tool can be integrated into our build process. Adding `workbox-build` to our build process eliminates the need for manual updates to the precache manifest each time that we update the apps files.
2. `workbox-build` automatically adds "revision hashes" to the files in the manifest entries. The revision hashes enable Workbox to intelligently track when files have been modified or are outdated, and automatically keep caches up to date with the latest file versions. Workbox can also remove cached files that are no longer in the manifest, keeping the amount of data stored on a user's device to a minimum. You'll see what `workbox-build` and the file revision hashes look like in the next section.

Learn more

While we are using workbox-build with gulp in this lab, Workbox also supports tools like webpack with workbox-webpack-plugin and npm-build processed with workbox-cli.

3. Inject a manifest into the service worker

Now we need to configure `workbox-build` to inject a precache manifest in the `precacheAndRoute` call in the service worker file.

Add code to include the `workbox-build` module into **gulpfile.js**:

gulpfile.js

```
const workboxBuild = require('workbox-build');
```

Then add a `build-sw` task in **gulpfile.js** to generate and inject the precache manifest:

gulpfile.js

```
gulp.task('build-sw', () => {
  return workboxBuild.injectManifest({
    swSrc: 'src/sw.js',
    swDest: 'build/sw.js',
    globDirectory: 'build',
    globPatterns: [
      '<em>*\/</em>.css',
      'index.html',
      'js\/animation.js',
      'images\/home\/*.jpg',
      'images\/icon\/*.svg'
    ]
  }).catch(err => {
    console.log('Uh oh ', err);
  });
});
```

Finally, update the `default` gulp task to include the `build-sw` task in its `runSequence`. The updated `default` task should be as follows:

gulpfile.js

```
gulp.task('default', ['clean'], cb => {
  runSequence(
    'copy',
    'build-sw',
    cb
  );
});
```

Since we've updated **gulpfile.js**, we need to rerun the gulp serve task in the command line. Terminate the `gulp serve` process with `Ctrl+c` and then restart it with `gulp serve`.

Observe that the `precacheAndRoute` call in the production service worker (**build/sw.js**) has been updated with the precache manifest, and should look like this (your hash values may be different):

build/sw.js

```
workbox.precaching.precacheAndRoute([
  {
    "url": "style/main.css",
    "revision": "919792b0fc1138b73c7553f56cae2c41"
  },
  {
    "url": "index.html",
    "revision": "49dc015edffceba2a88e909b944675c0"
  },
  // ...
]);
```

Explanation

The `build-sw` task uses the `workbox-build` module's `injectManifest` method. This method copies a source service worker file, specified in `swSrc`, to an output specified by `swDest`. Workbox searches the new service worker for an empty `precacheAndRoute()` call and populates the empty array with the assets defined in `globPatterns`. Adding the `build-sw` task to the `default` task ensures that the production service worker is regenerated anytime the **src** files change.

Learn more

Precaching documentation

4. Register and test the service worker

Now add a script in the bottom of **index.html** (just before the closing `</body>` tag) to register the new service worker:

index.html

```
<script>
  if ('serviceWorker' in navigator) {
    window.addEventListener('load', () => {
      navigator.serviceWorker.register('/sw.js')
        .then(registration => {
          console.log('Service Worker registered! 😎');
        })
        .catch(err => {
          console.log('Registration failed 😢 ', err);
        });
    });
  }
</script>
```

Return to the app and unregister any existing service workers and clear all service worker caches. In Chrome DevTools, you can do this in one easy operation by going to the **Application** tab, clicking **Clear Storage** and then clicking the **Clear site data** button.

Tip: You can automatically activate updated service workers in Chrome Developer Tools by selecting **Update on reload** in the **Server Workers** tab.
Refresh the page and check the developer console; you should see the registration success message. You can also use developer tools to check that status of the service worker. In Chrome DevTools, you can find service workers in the **Service Worker** section of the **Application** tab.

Next, observe that the resources specified in the precache manifest have been cached. In Chrome DevTools, you can see your caches by expanding the **Cache Storage** section in the **Application** tab.

Note: In Chrome, the DevTools Cache interface is not always updated automatically, even if new files are cached. If you don't see the cached files, right-click **Cache Storage** in the **Application** tab and choose **Refresh Caches**. If a similar method is not available in your browser, then try reloading the page.
Now go "offline" by using `Ctrl+C` to terminate the gulp server. Refresh the app in the browser. The home page should load even though we are offline!

Note: In Chrome, you may see a console error indicating that the service worker could not be fetched: `An unknown error occurred when fetching the script. service-worker.js Failed to load resource: net::ERR_CONNECTION_REFUSED` . This error is shown because the browser

couldn't fetch the service worker script (because the site is offline), but that's expected because we can't use the service worker to cache itself. Otherwise the user's browser would be stuck with the same service worker forever!

Explanation

In addition to precaching, the `precacheAndRoute` method sets up an implicit cache-first handler, ensuring that the precached resources are served offline.

Note: Workbox also handles an edge case that we haven't mentioned - Workbox knows to serve **my-domain/index.html** even if **my-domain/** is requested! With this functionality, you don't have to manage multiple cached resources (one for **index.html** and one for **/**).

5. Add routes to the service worker

workbox-sw.js has a `rout ing` module that lets you easily add routes to your service worker.

Let's add a route to the service worker now. Copy the following code into **src/sw.js** beneath the `precacheAndRoute` method. Make sure you're not editing the production service worker, **build/sw.js**, as this file will be overwritten by our build process.

src/sw.js

```
workbox.routing.registerRoute(
  /(.<em>)articles(.</em>)\.(?:png|gif|jpg)/,
  workbox.strategies.cacheFirst({
    cacheName: 'images-cache',
    plugins: [
      new workbox.expiration.Plugin({
        maxEntries: 50,
        maxAgeSeconds: 30 * 24 * 60 * 60, // 30 Days
      })
    ]
  })
);
```

Save the file.

Rebuild the app (including the service worker) and restart the server with the following command:

```
gulp serve
```

Refresh the app and (activate the updated service worker) [tools_for_pwa_developers.md#activate] in the browser. In Chrome DevTools you can activate a new service worker by clicking **skipWaiting** in the **Service Worker** section of the **Application** tab. Navigate to **Article 1** and **Article 2** . Check the caches to see that the `images-cache` now exists and contains the images from Articles 1 and 2. You may need to refresh the caches in developer tools to see the contents.

Explanation

The `registerRoute` method on the `rout ing` class adds a route to the service worker. The first parameter in `registerRoute` is a regular expression URL pattern to match requests against. The second parameter is the handler that provides a response if the route matches. In this case the route uses the `strategies` class to access the `cacheFirst` run-time caching strategy. This means that whenever the app requests article images, the service worker checks the cache first for the resource before going to the network.

The handler in this code also configures Workbox to maintain a maximum of 50 images in the cache (ensuring that user's devices don't get filled with excessive images). Once 50 images has been reached, Workbox will remove the oldest image automatically. The images are also set to expire after 30 days, signaling to the service worker that the network should be used for those images.

Optional: Write your own route that caches the user avatar. The route should match requests to `/images/icon/*` and handle the request/response using the `staleWhileRevalidate` strategy. Give the cache the name "icon-cache" and allow a maximum of 5 entries to be stored in the cache. This strategy is good for icons and user avatars that change frequently but the latest versions are not essential to the user experience. You'll need to remove the icon from the precache manifest so that the service worker uses your `staleWhileRevalidate` route instead of the implicit cache-first route established by the `precache` method.

6. Use a customized networkFirst cache strategy

6.1 Write the basic handler using the `handle` method

Sometimes content must always be kept up-to-date (e.g., news articles, stock figures, etc.). For this kind of data, the `cacheFirst` strategy is not the best solution. Instead, we can use the `networkFirst` strategy to fetch the newest content first, and only if that fails does the service worker get old content from the cache.

Copy the following code below the existing route in **src/sw.js**:

src/sw.js

```
const articleHandler = workbox.strategies.networkFirst({
  cacheName: 'articles-cache',
  plugins: [
    new workbox.expiration.Plugin({
      maxEntries: 50,
    })
  ]
});

workbox.routing.registerRoute(/(.<em>)article(.</em>)\.html/, args => {
  return articleHandler.handle(args);
});
```

Save the file, refresh the app, and update the service worker in the browser. Reload the home page and click a link to one of the Trending Articles. Check the caches to see that the `articles-cache` was created and contains the article you just clicked. You may need to refresh the caches to see the changes.

Optional: Test that articles are cached dynamically by visiting some while online. Then take the app offline again by pressing `Ctrl+C` in the command line and re-visit those articles. Instead of the browser's default offline page, you should see the cached article. Remember to re-run `gulp serve` to restart the server.

Optional: Test the `networkFirst` strategy by changing the text of the cached article and reloading the page. Even though the old article is cached, the new one is served and the cache is updated.

Explanation

Here we are using the `networkFirst` strategy to handle a resource we expect to update frequently (trending news articles). This strategy updates the cache with the newest content each time it's fetched from the network.

Unlike the images route established in the previous step, the code above uses the `handle` method on the built-in `networkFirst` strategy. The `handle` method takes the object passed to the handler function (in this case we called it `args`) and returns a promise that resolves with a response. We could have passed in the caching strategy directly to the second argument of `registerRoute` as we did in the previous example, but instead we return a call to the `handle` method in a custom handler function. This technique gives us access to the response, as you'll see in the next step.

6.2 Handle invalid responses

The `handle` method returns a promise resolving with the response, so we can access the response with a `.then`.

Add the following `.then` inside the `article` route after the call to the `handle` method:

sw.js

```
.then(response => {
    if (!response) {
      return caches.match('pages/offline.html');
    } else if (response.status === 404) {
      return caches.match('pages/404.html');
    }
    return response;
  });
```

Then add **pages/offline.html** and **pages/404.html** to the `globPatterns` in the `build-sw` task defined in **gulpfile.js**. The updated `build-sw` task should look like this:

workbox-config.js

```
gulp.task('build-sw', () => {
  return workboxBuild.injectManifest({
    swSrc: 'src/sw.js',
    swDest: 'build/sw.js',
    globDirectory: 'build',
    globPatterns: [
      '<em>*\/</em>.css',
      'index.html',
      'js\/animation.js',
      'images\/home\/*.jpg',
      'images\/icon\/*.svg',
      'pages/offline.html',
      'pages/404.html'
    ]
  }).catch(err => {
    console.log('Uh oh ', err);
  });
});
```

Save the files. Restart gulp by terminating the `serve` process with `Ctrl+c` and re-running `gulp serve`. Refresh the app activate the updated service worker in the browser. On the app home page, try clicking the **Non-existent article** link. This link points to an HTML page,

pages/article-missing.html, that doesn't actually exist. You should see the custom 404 page that we precached!

Now try taking the app offline by pressing `Ctrl+C` in the command line and then click any of the links to unvisited articles. You should see the custom offline page!

Note: If you've already cached all the articles, you can clear the caches and refresh the page in order to test the offline page.

Explanation

The `.then` statement receives the response passed in from the handle method. If the response doesn't exist, then it means the user is offline and the response was not previously cached. Instead of letting the browser show a default offline page, the service worker returns the custom offline page that was precached. If the response exists but the status is 404, then our custom 404 page is returned. Otherwise, we return the original response.

7. Optional: Add a customized cacheFirst cache strategy

So far you have implemented many caching strategies with Workbox, and in the previous section you learned how to add custom logic to the default Workbox caching options.

This last exercise is a challenge with less guidance (you can still see the solution code if you get stuck). You need to:

- Add a final service worker route for the app's "post" pages (**pages/post1.html**, **pages/post2.html**, etc.).
- The route should use Workbox's `cacheFirst` strategy, creating a cache called "posts-cache" that stores a maximum of 50 entries.
- If the cache or network returns a resources with a 404 status, then return the **pages/404.html** resource.
- If the resource is not available in the cache, and the app is offline, then return the **pages/offline.html** resource.

Hints

- Review the previous section for how to add additional logic to Workbox's built-in strategy handlers.
- The Workbox `cacheFirst` handler handles no-connectivity differently than `networkFirst`: instead of passing an empty response to the `.then`, it *rejects* and goes

to the next `.catch`.

Congratulations!

You have learned how to use Workbox to create production-ready service workers!

What we've covered

- Writing a service worker using the **workbox-sw.js** library
- Adding routes to your service worker using **workbox-sw.js**
- Using the predefined caching strategies provided in **workbox-sw.js**
- Augmenting the **workbox-sw.js** caching strategies with custom logic
- Generating a production-grade service worker with `workbox-build`

Resources

- Workboxjs.org
- Workbox - developers.google.com

Lab: Migrating to Workbox from sw-precache and sw-toolbox

Content

Overview

1. Get set up

2. Explore the app

3. Test the app

4. Install workbox-sw and workbox-build

5. Write the service worker using workbox-sw

6. Replace the sw-toolbox routes with Workbox routes

7. Precache the static assets

Congratulations!

Overview

Workbox is the successor to `sw-precache` and `sw-toolbox`. It is a collection of libraries and tools used for generating a service worker, precaching, routing, and runtime-caching. Workbox also includes modules for easily integrating background sync and offline Google Analytics into your service worker. See the Workbox page on developers.google.com for an explanation of each module contained in Workbox.

This lab shows you how to take an existing PWA that uses `sw-precache` and `sw-toolbox` and migrate it to Workbox to create optimal service worker code. This lab may only be useful to you if you have an existing PWA that was written with sw-precache and sw-toolbox. If you want to learn how to use Workbox from scratch, see this lab.

What you will learn

- The Workbox equivalents of `sw-precache` and `sw-toolbox` strategies
- How to write a service worker using the `workbox-sw` library

- How to add routes to your service worker using `workbox-sw`
- How to use the predefined caching strategies provided in `workbox-sw`
- How to inject a manifest into your service worker using `workbox-build`

What you should already know

- Basic HTML, CSS, and JavaScript
- ES2015 Promises
- How to run commands from the command line
- Familiarity with gulp
- Experience with sw-toolbox/sw-precache

What you will need

- Computer with terminal/shell access
- Connection to the internet
- A browser that supports service worker
- A text editor
- Node and npm

1. Get set up

If you have not already downloaded the repository, follow the instructions in Setting up the labs. You don't need to start the server for this lab.

In **sw-precache-workbox-lab/**, open the **project** folder in your preferred text editor. The **project** folder is where you do the work in this lab.

2. Explore the app

The starting app is a working PWA that currently uses `sw-precache` and `sw-toolbox` to generate a service worker and manage its routes. Open **project/gulpfile.js** and look at the code. Notice the `service-worker` task uses `sw-precache` to generate a service worker and imports the `sw-toolbox` library (**sw-toolbox.js**) and a custom `sw-toolbox` script (**js/toolbox-script.js**) into the generated service worker:

gulpfile.js

```
gulp.task('service-worker', function(callback) {
  swPrecache.write('build/sw.js', {
    staticFileGlobs: [
      'build/index.html',
      'build/css/main.css',
    ],
    importScripts: [
      'sw-toolbox.js',
      'js/toolbox-script.js'
    ],
    stripPrefix: 'build'
  }, callback);
});
```

This lab shows you how to translate this code so that it uses `workbox-build`, which is the Workbox version of `sw-precache`.

Let's look at the custom `sw-toolbox` script now. Open **app/js/toolbox-script.js** and look at the code. The file contains a couple routes that use the `cacheFirst` strategy to handle requests for Google fonts and images, and puts them into caches named `googleapis` and `images`, respectively:

```
toolbox.router.get('/(.*)', toolbox.cacheFirst, {
  cache: {
    name: 'googleapis',
    maxEntries: 20,
  },
  origin: /\.googleapis\.com$/
});

toolbox.router.get(/\.(?:png|gif|jpg)$/, toolbox.cacheFirst, {
  cache: {
    name: 'images',
    maxEntries: 50
  }
});
```

In the following steps, we'll replace these routes with Workbox routes using the `workbox-sw` library.

3. Test the app

Now that you have seen the code, run the app so you can see how it works.

Run the following command in the **project** directory to install the project dependencies:

```
npm install
```

After it's finished installing, run the gulp task to start the app:

```
gulp serve
```

This copies all of the relevant files to a build directory, generates a service worker (**build/sw.js**), starts a server, and opens the app in the browser.

After the app opens in the browser, open your browser's Developer Tools and verify that the service worker was installed. Then, open the cache and verify that the **index.html** and **main.css** files are cached.

Refresh the page and then refresh the cache and verify that the `googleapis` and `images` caches were created and they contain the font and image assets. Now let's convert the app so that we get the same results using Workbox.

4. Install `workbox-sw` and `workbox-build`

Close the app in the browser and stop the server with `ctrl+c`.

After the server has stopped, run the following command in the **project** directory to remove the `node_modules` folder containing the `sw-precache` and `sw-toolbox` modules:

```
rm -rf node_modules
```

Next, install the `workbox-sw` and `workbox-build` modules:

```
npm install --save workbox-sw
npm install --save-dev workbox-build
```

Then, open the **package.json** file and delete `sw-toolbox` from the `dependencies` and delete `sw-precache` from the `devDependencies`. The full `package.json` file should look like the following (your version numbers may differ):

package.json

```
{
  "name": "responsive-blog",
  "version": "1.0.0",
  "description": "",
  "main": "gulpfile.js",
  "dependencies": {
    "workbox-sw": "^2.0.0"
  },
  "devDependencies": {
    "browser-sync": "^2.18.13",
    "del": "^2.2.2",
    "gulp": "^3.9.1",
    "run-sequence": "^1.2.2",
    "workbox-build": "^2.1.0"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Save the file.

Finally, reinstall the project dependencies:

```
npm install
```

Explanation

In this step, we install the necessary Workbox libraries and remove the unused `sw-precache` and `sw-toolbox` libraries. `workbox-sw` is a high-level library that contains methods to add routes to your service worker, and a method to precache assets. This module replaces `sw-toolbox`, although it is not a one-to-one mapping. We'll explore the differences later in the lab.

`workbox-build` </code> replaces `sw-precache` for gulp applications. Workbox also contains build modules for webpack and the command line, which are the `workbox-webpack-plugin` and `workbox-cli`, respectively.

5. Write the service worker using `workbox-sw`

With `sw-precache` and `sw-toolbox`, the process of writing a service worker went something like this:

1. Write the routes in a separate file (in this case, **js/toolbox-script.js**) using `sw-toolbox`
2. Generate the service worker using `sw-precache` and import the file containing the routes

This generated a service worker file that contained all of the code in the `sw-precache` library.

With Workbox, the process goes something like this:

1. Write the service worker file itself using `workbox-sw`
2. Inject the files you want to precache into the service worker file using `workbox-build`

Workbox lets you write the service worker file yourself and provides helper modules for doing common tasks in the service worker. The most popular approach is to use the high-level `workbox-sw` module to write the service worker, which contains methods for precaching assets, routing, and performing different caching strategies. After the service worker is written, you can use one of the Workbox build tools, like `workbox-build`, to inject a list of files you want to precache (known as a precache manifest) into the service worker. This process results in a service worker file that is more readable and easier to customize.

Let's write the service worker now. Create an **sw.js** file in **app/** and add the following snippet to it:

app/sw.js

```
importScripts('workbox-sw.dev.v2.1.0.js');

const workboxSW = new WorkboxSW();
workboxSW.precache([]);
```

Now copy the `workbox-sw` file at **node_modules/workbox-sw/build/importScripts/workbox-sw.dev.v2.1.0.js** and paste it into **app/**.

Note: If you have a newer workbox-sw version, remember to update the version number in the importScripts call in your service worker.

Explanation

The `workbox-sw` module exposes a few methods you can use to write the service worker.

The `precache` method takes a file manifest and caches the assets on service worker install. Note that we've left the array empty: It is recommended that you populate this array using a Workbox build module, such as `workbox-build`. We'll look at how and why in step 7.

See the documentation for full descriptions of the Workbox modules and methods.

6. Replace the sw-toolbox routes with Workbox routes

With `workbox-sw`, we can write the routes directly in the service worker, instead of importing a separate file like we did with `sw-toolbox`. This leaves us with one less file to serve and a simpler development process.

Append the following routes to **app/sw.js**. These are the Workbox equivalents of the existing `sw-toolbox` routes.

app/sw.js

```
workboxSW.router.registerRoute('https://fonts.googleapis.com/(.*)',
  workboxSW.strategies.cacheFirst({
    cacheName: 'googleapis',
    cacheExpiration: {
      maxEntries: 20
    },
    cacheableResponse: {statuses: [0, 200]}
  })
);

workboxSW.router.registerRoute(/\.(?:png|gif|jpg)$/,
  workboxSW.strategies.cacheFirst({
    cacheName: 'images',
    cacheExpiration: {
      maxEntries: 50
    }
  })
);
```

Save the file. You can delete the **js** folder containing the **toolbox-script.js** file and the **sw-toolbox.js** file in **app/**.

Explanation

We've translated the `sw-toolbox` routes into Workbox routes, and put them directly in the service worker. See the Router documentation for more information on Workbox routes.

7. Precache the static assets

So far, we've removed the `sw-precache` and `sw-toolbox` modules and replaced them with `workbox-build` and `workbox-sw`. We've written the new service worker and translated the `sw-toolbox` routes to `workbox-sw` routes. The only thing left to do is to translate the `sw-precache` gulp task into a `workbox-build` task to inject the manifest into the service worker. Let's do that now.

In **app/gulpfile.js**, replace the line requiring `sw-precache` with the following line to require the `workbox-build` module:

app/gulpfile.js

```
const wbBuild = require('workbox-build');
```

Then, replace the `service-worker` task in **app/gulpfile.js** with the following task:

app/gulpfile.js

```
gulp.task('service-worker', () => {
  return wbBuild.injectManifest({
    swSrc: 'app/sw.js',
    swDest: 'build/sw.js',
    globDirectory: 'build',
    staticFileGlobs: [
      'index.html',
      'css/main.css'
    ]
  })
  .catch((err) => {
    console.log('[ERROR] This happened: ' + err);
  });
});
```

Save the file.

Before testing your changes, make sure you've closed all open instances of the app in the browser. The service worker won't update if any of the pages it controls are still open. Then, start the server in the **project** directory with `gulp serve`.

After the app opens in the browser, open Developer Tools and unregister the previous service worker and clear the caches. Refresh the page a couple times so the new service worker can install and intercept some network requests. Check that the `workbox-precaching-revisioned` cache exists and contains **index.html** and **css/main.css**. Check that the `googleapis` and `images` caches were created and contain the appropriate files. If everything is there you've successfully migrated the app to Workbox!

Explanation

The `injectManifest` method copies the source service worker file to the destination service worker file, searches the new service worker for an empty `precache()` call (such as `.precache([])`), and populates the empty array with the assets defined in `staticFileGlobs`. It also creates hashes of these files so that Workbox can intelligently update the caches if you change any of the files.

Congratulations!

You have learned how to convert an app that uses `sw-precache` and `sw-toolbox` to one that uses Workbox!

What we've covered

- Using `workbox-sw` to precache static assets
- Using `workbox-sw` to create routes in your service worker
- Using `workbox-build` to inject a list of files for your service worker to precache

Resources

- Workboxjs.org
- Workbox - developers.google.com

Lab: Integrating Web Push

Contents

Overview

1. Get set up

2. Using the Notification API

3. Using the Push API

4. Optional: Identifying your service with VAPID

5. Optional: Best practices

Congratulations!

Concepts: Introduction to Push Notifications

Overview

This lab shows you the basics of sending, receiving, and displaying push notifications. Notifications are messages that display on a user's device, outside of the context of the browser or app. Push notifications are notifications created in response to a message from a server, 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.

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
- How to design push notifications into your app following best practices

What you should know

- Have completed the Service Worker course or have equivalent experience with Service Worker

- Have completed the Promises codelab or have equivalent experience
- Intermediate experience using the command line interface
- Intermediate-to-advanced experience with JavaScript

What you will need

- Computer with terminal/shell access
- Connection to the Internet
- A Google or Gmail account
- A browser that supports web push
- Node and npm

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

In the command window, change to the **app** directory in the **push-notification-lab** and run the following:

```
npm install
```

This reads the dependencies in **package.json** and installs the web-push module for Node.js, which we will use in the second half of the lab to push a message to our app.

Then install web-push globally so we can use it from the command line:

```
npm install web-push -g
```

Open your browser and navigate **localhost:8080/push-notification-lab/app/**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **push-notification-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **images** folder contains sample images
- **js/main.js** is the main JavaScript for the app, and where you will write the app's script
- **node/main.js** is the Node.js server

- **samples** folder contains sample landing pages
- **index.html** is the main HTML page for our sample site/application
- **manifest.json** is the Firebase manifest file
- **package.json** is the Node.js manifest file
- **sw.js** is the service worker file where we will write the script to handle notifications

2. Using the Notification API

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

2.1 Check for support

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

Replace TODO 2.1 in **main.js** with the following code:

main.js

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

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

2.2 Request permission

Before we can show notifications, we must get permission from the user.

Replace TODO 2.2 in **main.js** with the following code:

main.js

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

Let's test this function in the browser. Save the code and refresh the page in the browser. A message box should appear at the top of the browser window prompting you to allow notifications.

If the prompt does not appear, you can set the permissions manually by clicking the **Information** icon in the URL bar. As an experiment, try rejecting permission 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 a notification. This permission status is stored in the browser, so calling this again returns the user's last choice.

2.3 Display the notification

Replace TODO 2.3 in **main.js** in the `displayNotification()` function with the following code:

main.js

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

    // TODO 2.4 - Add 'options' object to configure the notification

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

Save the file and reload the page in the browser. 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

2.4 Add notification options

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

Replace TODO 2.4 in **main.js** with an options object:

main.js

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

  // TODO 2.5 - add actions to the notification

  // TODO 5.1 - add a tag 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. Click **Notify me!** In the browser to see the new additions to the notification.

Explanation

`showNotification` has an optional second parameter that 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 lets your app 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.

2.5 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 2.5 in the options object in **main.js** with the following code:

main.js

```
actions: [
  {action: 'explore', title: 'Go to the site',
    icon: 'images/checkmark.png'},
  {action: 'close', title: 'Close the notification',
    icon: 'images/xmark.png'},
]
```

Save the code and reload the page in the browser. Click **Notify me!** on the page to display a notification. The notification now has two new buttons to click (these are not available in Firefox). These don't do anything yet. In the next sections we'll 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.

2.6 Handle the notificationclose event

When the user closes a notification, a `notificationclose` event is triggered in the service worker.

Replace TODO 2.6 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);
});
```

Save the code and update the service worker in the browser. Now, in the page, click **Notify 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.

Tip: The `notificationclose` event is a great place to add Google analytics to see how often users are closing our notifications. You can learn more about this in the Google Analytics codelab.

2.7 Handle the notificationclick event

When the user clicks on a notification or notification action, a `notificationclick` event is triggered in the service worker.

Replace the TODO 2.7 in **sw.js** with the following code:

sw.js

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

  // TODO 2.8 - change the code to open a custom page

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

Save the code and reload the page.Update the service worker in the browser. Click **Notify me!** to create a new notification and click it. You should land on the Google homepage.

2.8 Optional: Open a custom page from the notification

To complete TODO 2.8 inside the `notificationclick` event, write the code to complete the following tasks. The first two steps will be the same as the first two lines of code in the `notificationclose` event handler.

1. Get the notification from the event object and assign it to a variable called "notification".
2. Then get the `primaryKey` from the data in the notification and assign it to a `primaryKey` variable.
3. Replace the URL in `clients.openWindow` with `'samples/page' + primaryKey + '.html'`.
4. 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.

Save the code and update the service worker in the browser. Click **Notify me!** to create a new notification and then 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. This should take you to **page2.html** when you click the notification.

2.9 Handle actions

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

Replace the entire `notificationclick` event listener in **sw.js** with the following code:

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();
  }

  // TODO 5.3 - close all notifications when one is clicked

});
```

Save the code and update the service worker in the browser. Click **Notify me!** to create a new notification. Try clicking the actions.

Note: Notice we check for the "close" action first and handle the "explore" action in an `else` block. This is a best practice as not every platform supports action buttons, and not every platform displays all your actions. Handling actions in this way provides a default experience that works everywhere.

Solution code

The solution code can be found in the **02-9-handle-events** directory.

3. Using the Push API

The Push API is an interface that lets your app subscribe to a push service and enables the service worker to receive push messages.

For more information

- Push API - MDN
- Using the Push API - MDN

3.1 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 3.1 with the code to handle push events:

sw.js

```
self.addEventListener('push', function(e) {
  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: 'Go to the site',
        icon: 'images/checkmark.png'},
      {action: 'close', title: 'Close the notification',
        icon: 'images/xmark.png'},
    ]
  };
  e.waitUntil(
    self.registration.showNotification('Hello world!', options)
  );
});
```

Save the code and update the service worker. Try sending a push message from the browser to your service worker. A notification should appear on your screen.

Note: Push notifications are currently only supported in Chrome and Firefox. See the entry for "push" on caniuse.com for the latest browser support status.

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.

For more information

- Push Event - MDN

3.2 Create a project on Firebase

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

If you are using Firefox, you can skip this step and continue to step 3.3.
Note: Recent changes to Firebase Cloud Messaging let developers avoid creating a Firebase account if the VAPID protocol is used. See the section on VAPID for more information.

1. In the Firebase console, select **Create New Project**.
2. Supply a project name and click **Create Project**.
3. Click the **Settings** icon (next to your project name in the Navigation panel), and select **Project Settings**.
4. Open 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"
}
```

Explanation

Chrome uses Firebase Cloud Messaging (FCM) to route its push messages. All push messages are sent to FCM, and then FCM passes them to the correct client.

Note: FCM has replaced Google Cloud Messaging (GCM). Some of the code to push messages to Chrome still contains references to GCM. These references are correct and work for both GCM and FCM.

3.3 Get the subscription object

Whenever the user opens the app, check for the subscription object and update the server and UI.

Replace TODO 3.3a in the service worker registration code at the bottom of **main.js** with the following function call:

main.js

```
initializeUI();
```

Replace TODO 3.3b in the `initializeUI()` function in **main.js** with the following code:

main.js

```
pushButton.addEventListener('click', function() {
  pushButton.disabled = true;
  if (isSubscribed) {
    unsubscribeUser();
  } else {
    subscribeUser();
  }
});

swRegistration.pushManager.getSubscription()
.then(function(subscription) {
  isSubscribed = (subscription !== null);

  updateSubscriptionOnServer(subscription);

  if (isSubscribed) {
    console.log('User IS subscribed.');
  } else {
    console.log('User is NOT subscribed.');
  }

  updateBtn();
});
```

Save the code.

Explanation

Here we add a click event listener to the **Enable Push Messaging** button in the page. The button calls `unsubscribeUser()` if the user is already subscribed, and `subscribeUser()` if they are not yet subscribed.

We then get the latest subscription object from the `pushManager`. In a production app, this is where we would update the subscription object for this user on the server. For the purposes of this lab, `updateSubscriptionOnServer()` simply posts the subscription object to the page so we can use it later. `updateBtn()` updates the text content of the **Enable Push Messaging** button to reflect the current subscription status. You'll need to use these functions later, so make sure you understand them before continuing.

3.4 Subscribe to the push service

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

Replace TODO 3.4 in **main.js** with the following code:

```
swRegistration.pushManager.subscribe({
  userVisibleOnly: true
})
.then(function(subscription) {
  console.log('User is subscribed:', subscription);

  updateSubscriptionOnServer(subscription);

  isSubscribed = true;

  updateBtn();
})
.catch(function(err) {
  if (Notification.permission === 'denied') {
    console.warn('Permission for notifications was denied');
  } else {
    console.error('Failed to subscribe the user: ', err);
  }
  updateBtn();
});
```

Save the code and refresh the page. Click **Enable Push Messaging**. The subscription object should display on the page. The subscription object contains the endpoint URL, which is where we send the push messages for that user, and the keys needed to encrypt the message payload. We use these in the next sections to send a push message.

Explanation

Here we subscribe to the `pushManager`. In production, we would then update the subscription object on the server.

The `.catch` handles the case in which the user has denied permission for notifications. We might then update our app with some logic to send messages to the user in some other way.

Note: We are setting the `userVisibleOnly` option to `true` in the subscribe method. By setting this to `true`, we ensure that every incoming message has a matching notification. The default setting is `false`. Setting this option to `true` is required in Chrome.

For more information

- PushSubscription - MDN
- Subscribe method - MDN
- Notification permission status - MDN

3.5 Unsubscribe from the push service

Replace TODO 3.5 in **main.js** with the following code:

```
swRegistration.pushManager.getSubscription()
.then(function(subscription) {
  if (subscription) {
    return subscription.unsubscribe();
  }
})
.catch(function(error) {
  console.log('Error unsubscribing', error);
})
.then(function() {
  updateSubscriptionOnServer(null);

  console.log('User is unsubscribed');
  isSubscribed = false;

  updateBtn();
});
```

Save the code and refresh the page in the browser. Click **Disable Push Messaging** in the page. The subscription object should disappear and the console should display `User is unsubscribed`.

Explanation

Here we unsubscribe from the push service and then "update the server" with a `null` subscription object. We then update the page UI to show that the user is no longer subscribed to push notifications.

For more information

- Unsubscribe method - MDN

3.6 Optional: Send your first web push message using cURL

Let's use cURL to test pushing a message to our app.

Note: Windows machines do not come with cURL preinstalled. If you are using Windows, you can skip this step.
In the browser, click **Enable Push Messaging** and copy the endpoint URL. Replace `ENDPOINT_URL` in the cURL command below with this endpoint URL.

If you are using Chrome, replace `SERVER_KEY` in the Authorization header with the server key you saved earlier.

Note: The Firebase Cloud Messaging server key can be found in your project on Firebase by clicking the Settings icon in the Navigation panel, clicking **Project settings** and then opening the **Cloud messaging** tab.
Paste the following cURL command (with your values substituted into the appropriate places) into a command window and execute:

```
curl "ENDPOINT_URL" --request POST --header "TTL: 60" --header "Content-Length: 0" --h
eader "Authorization: key=SERVER_KEY"
```

Here is an example of what the cURL should look like:

```
curl "https://android.googleapis.com/gcm/send/fYFVeJQJ2CY:APA91bGrFGRmy-sY6NaF8atX11K0
bKUUNXLVzkomGJFcP-lvne78UzYeE91IvWMxU2hBAUJkFlBVdYDkcwLG8vO8cYV0X3Wgvv6MbVodUfc0gls7HZ
cwJL4LFxjg0y0-ksEhKjpeFC5P" --request POST --header "TTL: 60" --header "Content-Length
: 0" --header "Authorization: key=AAAANVIuLLA:APA91bFVym0UAy836uQh-__S8sFDX0_MN38aZaxG
R2TsdbVgPeFxhZH0vXw_-E99y9UIczxPGHE1XC1CHXen5KPJlEASJ5bAnTUNMOzvrxsGuZFAX1_ZB-ejqBwaIo
24RUU5QQkLQb9IBUFwLKCvaUH9tzOl9mPhFw"
```

You can send a message to Firefox's push service by opening the app in Firefox, getting the endpoint URL, and executing the same cURL without the `Authorization` header.

Note: Remember to unregister the previous service worker at localhost if it exists.

```
curl "ENDPOINT_URL" --request POST --header "TTL: 60" --header "Content-Length: 0"
```

That's it! We have sent our very first push message. A notification should have popped up on your screen.

Explanation

We are using the Web Push protocol to send a push message to the endpoint URL, which contains the address for the browser's Push Service and the information needed for the push service to send the push message to the right client. For Firebase Cloud Messaging specifically, we must include the Firebase Cloud Messaging server key in a header (when not using VAPID). We do not need to encrypt a message that doesn't contain a payload.

For more information

- Push Demo
- Getting Started with cURL
- cURL Documentation

3.7 Get data from the push message

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

Replace the `push` event listener in **sw.js** with the following code to get the data from the message:

sw.js

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

  if (e.data) {
    body = e.data.text();
  } else {
    body = 'Default body';
  }

  var options = {
    body: body,
    icon: 'images/notification-flat.png',
    vibrate: [100, 50, 100],
    data: {
      dateOfArrival: Date.now(),
      primaryKey: 1
    },
    actions: [
      {action: 'explore', title: 'Go to the site',
        icon: 'images/checkmark.png'},
      {action: 'close', title: 'Close the notification',
        icon: 'images/xmark.png'},
    ]
  };

  e.waitUntil(
    self.registration.showNotification('Push Notification', options)
  );
});
```

Save the code.

Explanation

In this example, we're getting the data payload as text and setting it as the body of the notification.

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

3.8 Push the message from a Node.js server

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 3.8 in **node/main.js** with the code below.

Make sure you save the changes you made to the service worker in the last step and then unregister the service worker and refresh the page in the browser. Click **Enable Push Messaging** and copy the whole subscription object. Replace `YOUR_SUBSCRIPTION_OBJECT` in the code you just pasted into **node/main.js** with the subscription object.

If you are working in Chrome, replace `YOUR_SERVER_KEY` in the `options` object with your own Server Key from your project on Firebase. Do not overwrite the single quotes.

If you are working in Firefox, you can delete the `gcmAPIKey` option.

node/main.js

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

var pushSubscription = YOUR_SUBSCRIPTION_OBJECT;

// TODO 4.3a - include VAPID keys

var payload = 'Here is a payload!';

var options = {
  gcmAPIKey: 'YOUR_SERVER_KEY',
  TTL: 60,

  // TODO 4.3b - add VAPID details

};

webPush.sendNotification(
  pushSubscription,
  payload,
  options
);
```

Save the code. From the **push-notification-lab/app** directory, run the command below:

```
node node/main.js
```

A push notification should pop up on the screen. It may take a few seconds to appear.

Explanation

We are using the web-push Mozilla library for Node.js to simplify the syntax for sending a message to the push service. This library takes care of encrypting the message with the public encryption key. The code we added to **node/main.js** sets the Server key. It then passes the subscription endpoint to the `sendNotification` method and passes the public keys and payload to the object in the second argument.

For more information

- web-push library documentation
- Other Web Push libraries

Solution code

The solution code can be found in the **03-8-payload** directory.

4. Optional: Identifying your service with VAPID

Let's use the VAPID protocol to identify the app for the push service. This eliminates the need for a Firebase account.

4.1 Generate the keys

First, generate the public and private keys by entering the following command into a command window at the project directory:

```
web-push generate-vapid-keys [--json]
```

This generates a public/private key pair. The output should look like this:

```
=======================================

Public Key:
BAdXhdGDgXJeJadxabiFhmlTyF17HrCsfyIj3XEhg1j-RmT2wXU3lHiBqPSKSotvtfejZlAaPywJ9E-7AxXQBj
4

Private Key:
VCgMIYe2BnuNA4iCfR94hA6pLPT3u3ES1n1xOTrmyLw

=======================================
```

Copy your keys and save them somewhere safe. Use these keys for all future messages you send.

Note: The keys are URL Safe Base64 encoded strings.

4.2 Subscribe with the public key

In order for VAPID to work we must pass the public key to the `subscribe` method as a `Uint8Array`. We have included a helper function to convert the public key to this format.

Replace TODO 4.2a in **js/main.js**, with the following code with your VAPID public key substituted in:

js/main.js

```
var applicationServerPublicKey = 'YOUR_VAPID_PUBLIC_KEY';
```

Replace the `subscribeUser()` function in **js/main.js** with the code below:

js/main.js

```
function subscribeUser() {
  var applicationServerKey = urlB64ToUint8Array(applicationServerPublicKey);
  swRegistration.pushManager.subscribe({
    userVisibleOnly: true,
    applicationServerKey: applicationServerKey
  })
  .then(function(subscription) {
    console.log('User is subscribed:', subscription);
    updateSubscriptionOnServer(subscription);
    isSubscribed = true;
    updateBtn();
  })
  .catch(function(err) {
    if (Notification.permission === 'denied') {
      console.warn('Permission for notifications was denied');
    } else {
      console.error('Failed to subscribe the user: ', err);
    }
    updateBtn();
  });
}
```

Save the code. In the browser, click **Disable Push Messaging** or unregister the service worker. Then refresh the page and click **Enable Push Messaging**. If you are using Chrome, the endpoint URL domain should now be **fcm.googleapis.com**.

4.3 Sign and send the request

Copy the new subscription object and overwrite the old subscription object assigned to the `pushSubscription` variable in **node/main.js**.

Replace TODO 4.3a in **node/main.js** with the following code, with your values for the public and private keys substituted in:

node/main.js

```
var vapidPublicKey = 'YOUR_VAPID_PUBLIC_KEY';
var vapidPrivateKey = 'YOUR_VAPID_PRIVATE_KEY';
```

Next, replace TODO 4.3b in the `options` object with the following code containing the required details for the request signing:

Note: You'll need to replace `YOUR_EMAIL_ADDRESS` in the `subject` property with your actual email.

node/main.js

```
vapidDetails: {
  subject: 'mailto: YOUR_EMAIL_ADDRESS',
  publicKey: vapidPublicKey,
  privateKey: vapidPrivateKey
}
```

Comment out the `gcmAPIKey` in the options object (it's no longer necessary):

```
// gcmAPIKey: 'YOUR_SERVER_KEY',
```

Save the file. Enter the following command in a command window at the working directory (**push-notification-lab/app**):

```
node node/main.js
```

A push notification should pop up on the screen. It may take a few seconds to appear.

Note: The notification may not surface if you're in full screen mode.

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 Introduction to Web Push and the links below.

For more information

- Web Push Interoperability Wins
- Using VAPID

Solution code

The solution code can be found in the **04-3-vapid** directory.

5. Optional: Best practices

5.1 Manage the number of notifications

To complete TODO 5.1 in **main.js** in the `displayNotification` function, give the notification a `tag` attribute of `'id1'`.

Save the code and refresh the page in the browser. Click **Notify me!** multiple times. The notifications should replace 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

The solution code can be found in the **solution** directory.

5.2 When to show notifications

Depending on the use case, if the user is already using our application we may want to update the UI instead of sending them a notification.

In the `push` event handler in **sw.js**, replace the `e.waitUntil()` function below the TODO with the following code:

sw.js

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

Save the file and update the service worker, then refresh the page in the browser. Click **Enable Push Messaging**. Copy the subscription object and replace the old subscription object in **node/main.js** with it.

Execute the command to run the node server in the command window at the **app** directory:

```
node node/main.js
```

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 clients of the service worker on this machine. If there are no clients active, we create a notification.

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

Solution code

The solution code can be found in the **solution** directory.

5.3 Hide notifications on page focus

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

In **sw.js**, in the `notificationclick` event handler, replace the TODO 5.3 with the following code:

sw.js

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

Save the code.

Comment out the `tag` attribute in the `displayNotification` function in **main.js** so that multiple notifications will display at once:

main.js

```
// tag: 'id1',
```

Save the code, open the app again, and update the service worker. Click **Notify me!** a few times to display multiple notifications. If you click "Close the notification" on one notification they should all disappear.

Note: If you don't want to clear out all of the notifications, you can filter based on the `tag` attribute by passing the tag into the `getNotifications` function. See the getNotifications reference on MDN for more information.
Note: You can 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. We can clear out all of the notifications that we have created by iterating over the notifications returned from the `getNotifications` method on our service worker registration and then closing each notification.

Solution code

The solution code can be found in the **solution** directory.

5.4 Notifications and tabs

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

Replace the code inside the `else` block in the `notificationclick` handler in **sw.js** with the following code:

sw.js

```
e.waitUntil(
  clients.matchAll().then(function(clis) {
    var client = clis.find(function(c) {
      return 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();
    }
  })
);
```

Save the code and update the service worker in the browser. Click **Notify me!** to create a new notification. Try clicking on a notification once with your app open and focused, and once with a different tab open.

Note: The `clients.openWindow` method can only open a window when called as the result of a `notificationclick` event. Therefore, we need to wrap the method in a `waitUntil`, so that the event does not complete before `openWindow` is called. Otherwise, the browser throws an error.

Explanation

In this code we get all the clients of the service worker and assign the first "visible" client to the `client` variable. Then we open the page in this client. If there are no visible clients, we open the page in a new tab.

For more information

- openWindow() - MDN

Solution code

The solution code can be found in the **solution** directory.

Congratulations!

In this lab we have learned how to create notifications and configure them so that they work well and look great on the user's device. Push notifications are an incredibly powerful mechanism to keep in contact with your users. Using push notifications, it has never been easier to build meaningful relationships with your customer. By following the concepts in this lab you will be able to build a great experience that your users will keep coming back to.

What we've covered

- How to create notifications and configure them.
- How to build notifications that the user can interact with through either a single tap, or by clicking on one of a number of different actions.
- How to send messages to the user's device whether or not they have the browser open through the Open Web Push protocol, and how to implement this across all the browsers that support this API.

Resources

Introduction to push notifications

- Enable Push Notifications for your Web App
- Web Push Notifications: Timely, Relevant, and Precise

Demos

- Simple Push Demo
- Notification Generator

Learn about Web Push libraries

- Web Push Libraries

Learn about encryption

- Web Push Payload Encryption
- Web Push: Data Encryption Test Page

Learn about Firebase Cloud Messaging

- Firebase Cloud Messaging
- Set Up a JavaScript Firebase Cloud Messaging Client App

Lab: Integrating Analytics

Contents

Overview 1. Get set up 2. Create a Google Analytics account 3. Get your tracking ID and snippet 4. View user data 5. Use debug mode 6. Add custom events 7. Showing push notifications 8. Using analytics in the service worker 9. Use analytics offline 10. Optional: Add hits for notification actions 11. Optional: Use hitCallback Congratulations!

Concepts: Integrating Analytics

Overview

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

What you will learn

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

What you should know

- Basic JavaScript and HTML
- Familiarity with Push Notifications
- Some familiarity with the Fetch API
- The concept of an Immediately Invoked Function Expression (IIFE)
- How to enable the developer console

What you will need

- Computer with terminal/shell access
- Connection to the internet

- A browser that supports push
- A text editor
- Node installed

1. Get set up

If you have not downloaded the repository, installed Node, and started a local server, follow the instructions in Setting up the labs.

Open Chrome and navigate to **localhost:8080/google-analytics-lab/app**.

Note: Unregister any service workers and clear all service worker caches for localhost so that they do not interfere with the lab.
If you have a text editor that lets you open a project, open the **google-analytics-lab/app** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **app** folder is where you will be building the lab.

This folder contains:

- **pages** folder contains sample resources that we use in experimenting:
  - **page-push-notification.html**
  - **other.html**
- **images** folder contains images to style our notifications
- **index.html** is the main HTML page for our sample site/application
- **main.js** is the main JavaScript for the app
- **analytics-helper.js** is an empty helper file
- **sw.js** is the service worker file
- **manifest.json** is the manifest for push notifications
- **package.json** is used for tracking Node packages (optional)

In the browser, you should be prompted to allow notifications. If the prompt does not appear, then manually allow notifications. You should see a permission status of "granted" in the console.

You should also see that a service worker registration is logged to the console.

The app for this lab is a simple web page that has some push notification code. **main.js** requests notification permission and registers a service worker, **sw.js**. The service worker has listeners for push events and notification events.

main.js also contains functions for subscribing and unsubscribing for push notifications. We will address that later (subscribing to push isn't yet possible because we haven't registered with a push service).

Test the notification code by using developer tools to send a push notification.

A notification should appear on your screen. Try clicking it. It should take you to a sample page.

Note: The developer tools UI is constantly changing and, depending on the browser, may look a little different when you try it.
Note: Simulated push notifications can be sent from the browser even if the subscription object is null.

For more information

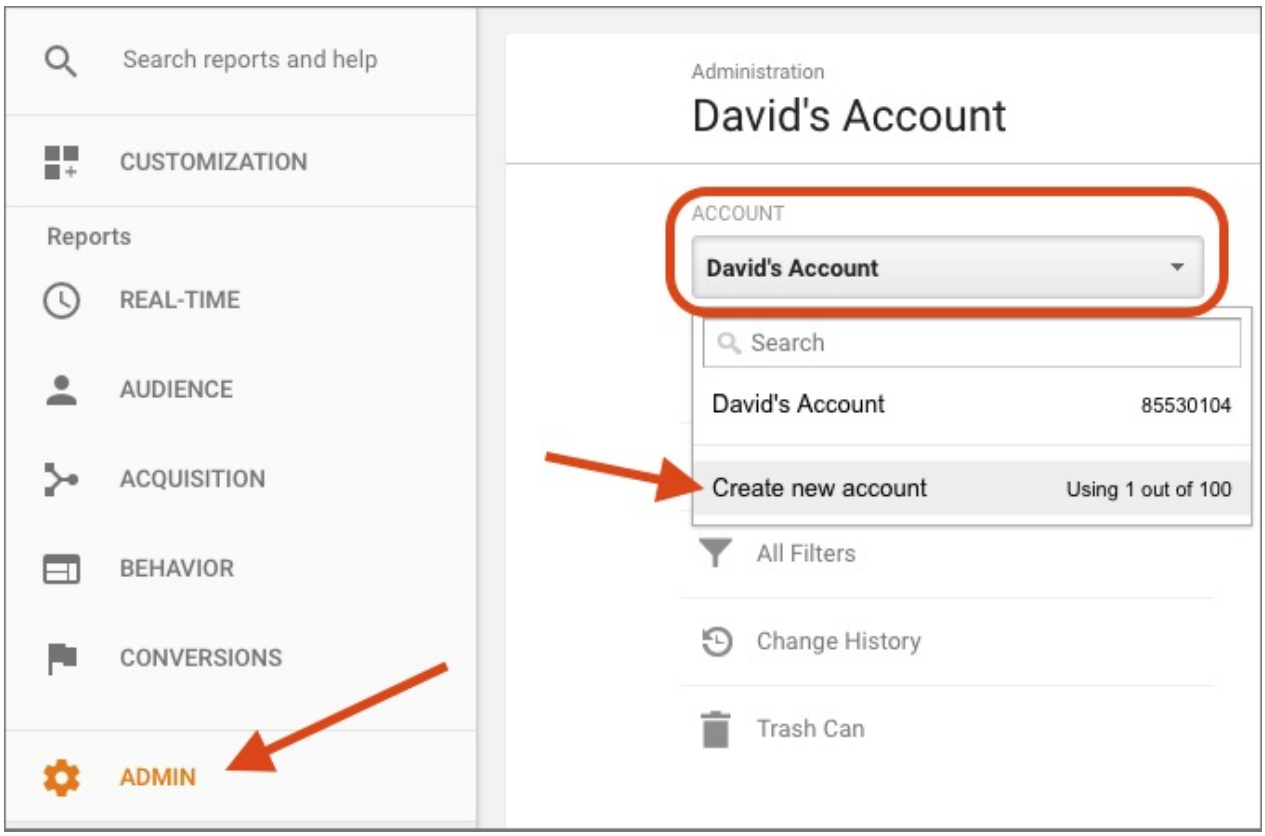
You can learn how to build the starter app and learn more about push in Push Notifications codelab.

2. Create a Google Analytics account

Note: The Google Analytics UI is subject to updates and may not look exactly like the screenshots presented in this lab.
In a separate tab or window, navigate to analytics.google.com. Sign in with your Gmail account, and follow the step that matches your status:

If you already have a Google Analytics account:

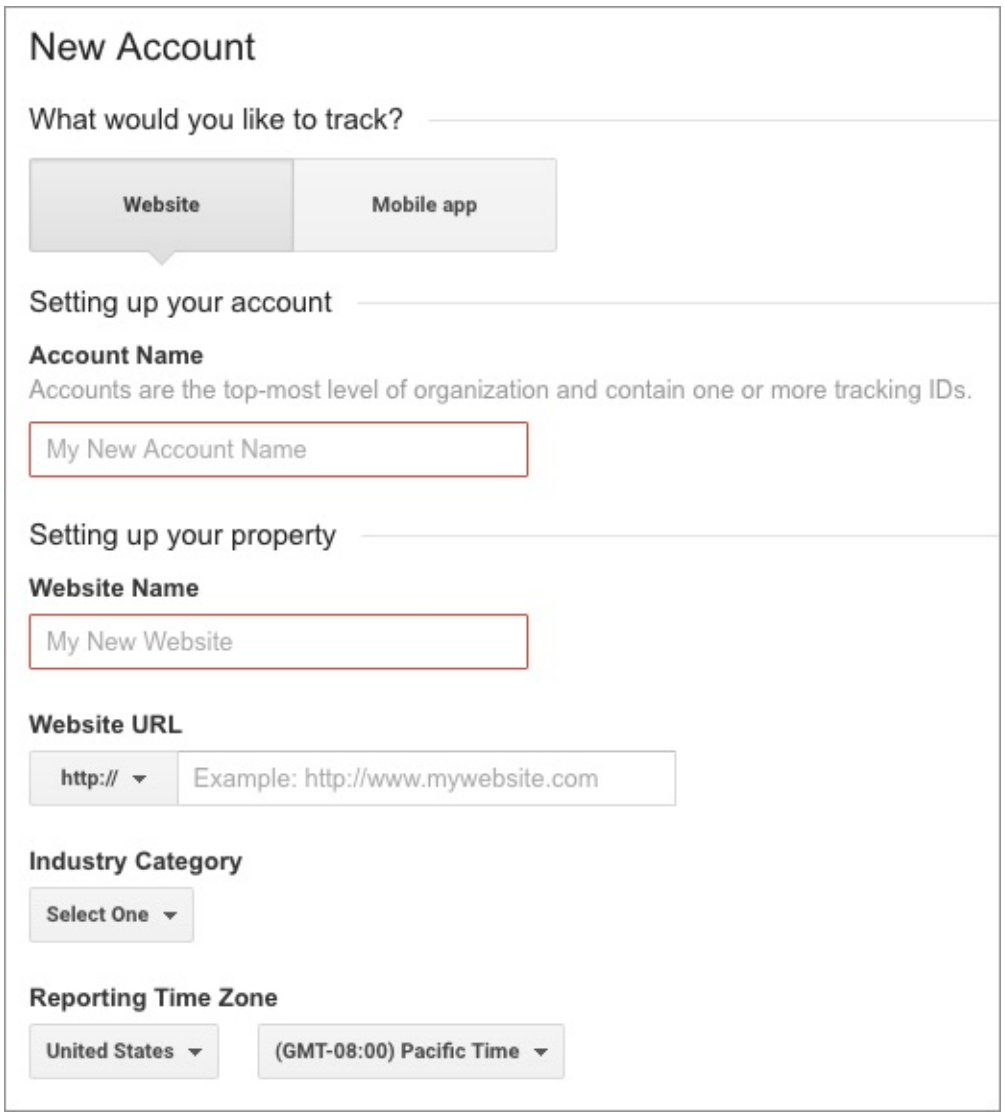
Create another one. Select the **Admin** tab. Under **account**, select your current Google Analytics account and choose **create new account**. A single Gmail account can have multiple (currently 100) Google Analytics accounts.



If you don't have a Google Analytics account:

Select **Sign up** to begin creating your account.

The account creation screen should look like this:



What would you like to track?

Choose website.

Note: Websites and mobile apps implement Google Analytics differently. This lab covers web sites. For mobile apps, see analytics for mobile applications.
Note: All the names we use for the account and website are arbitrary. They are only used for reference and don't affect analytics.

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 mock GitHub Pages site.

1. Set the website name to whatever you want, for example "GA Code Lab Site".
2. Set the website URL to **USERNAME.github.io/google-analytics-lab/**, where **USERNAME** is your GitHub username (or just your name if you don't have a GitHub account). Set the protocol to **https://**.

   Note: For this lab, the site is just a placeholder, you do not need to set up a GitHub Pages site or be familiar with GitHub Pages or even GitHub. The site URL that you use to create your Google Analytics account is only used for things like automated testing.
3. Select any industry or category.
4. Select your timezone.
5. Unselect any data sharing settings.
6. 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 IDs (also called property IDs) 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

3. Get your tracking ID and snippet

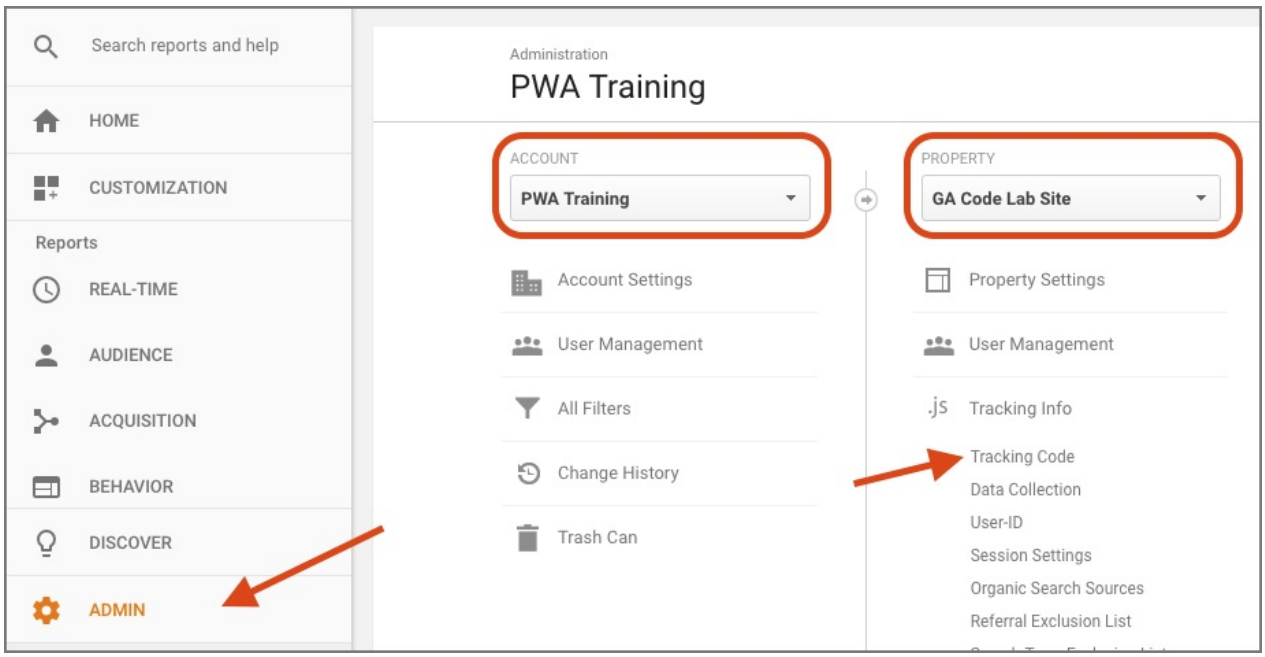
You should now see your property's tracking ID and tracking code snippet.

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 **property**, select your property (for example "GA Code Lab Site") from the

down list.

4. Now choose **Tracking Info**, followed by **Tracking Code**.



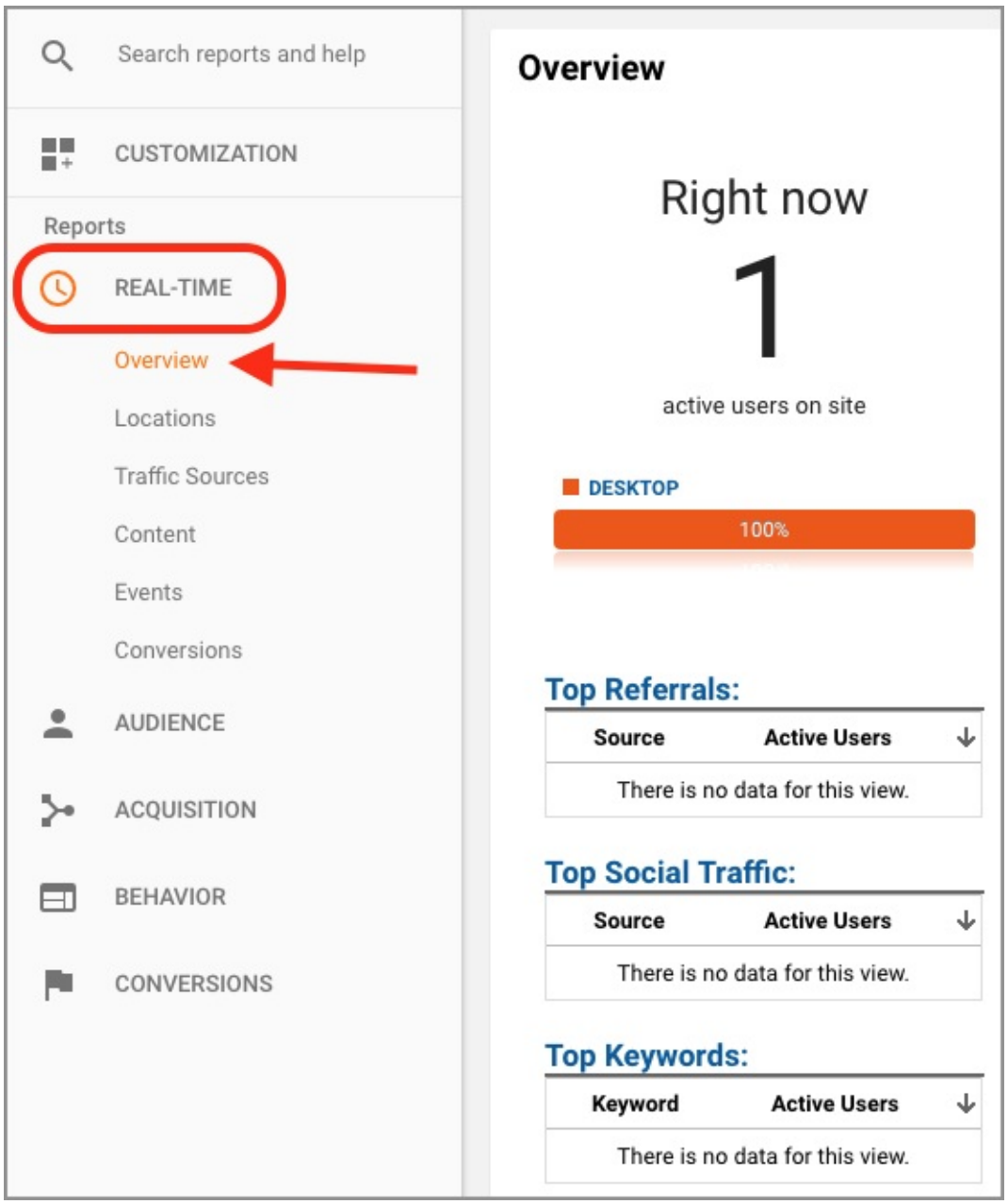
Your tracking ID looks like `UA-XXXXXXXX-Y` and your tracking code snippet looks like:

index.html

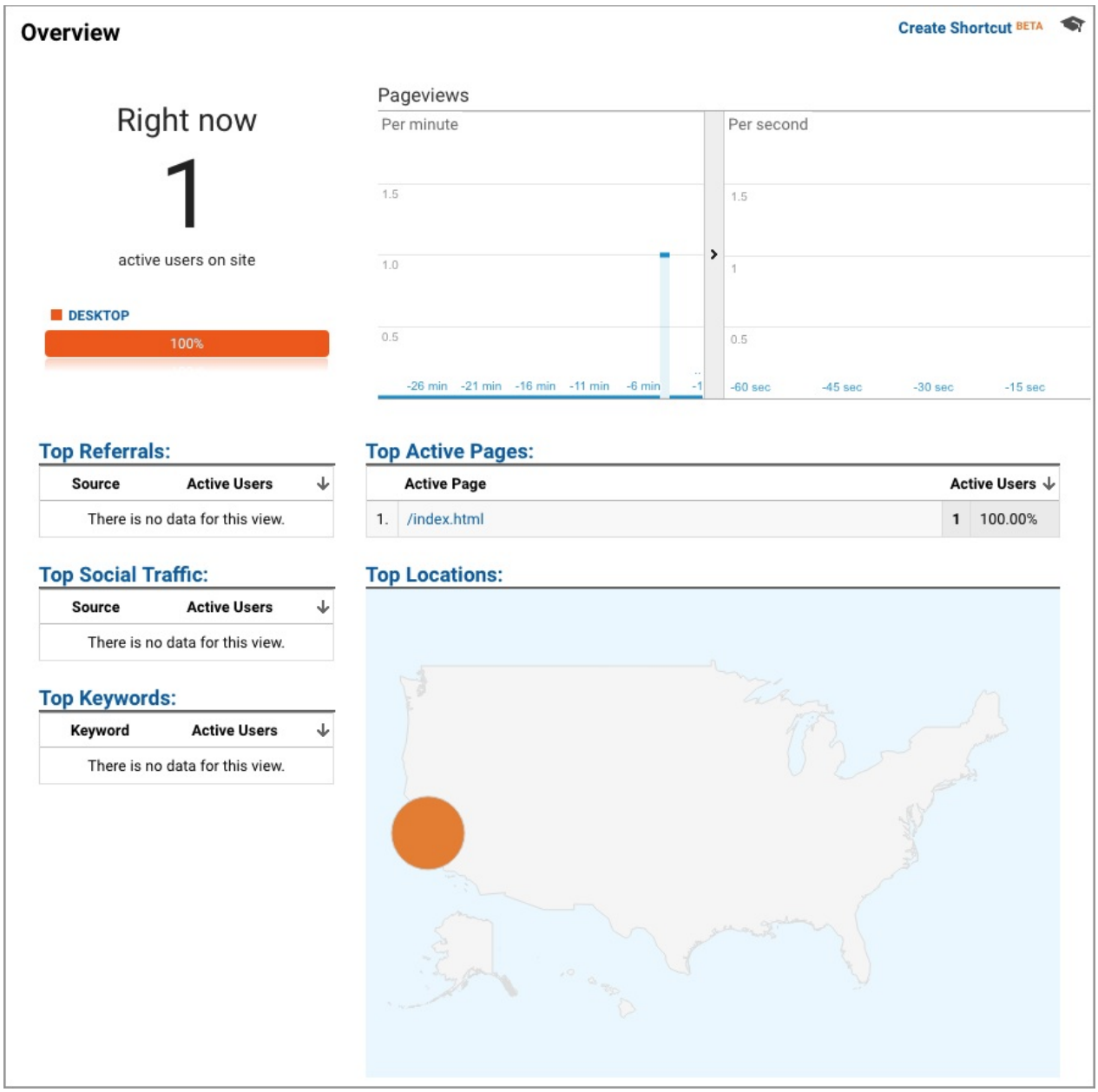
```
<script>
  (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){(i[r].q=
i[r].q||[]) \
.push(arguments)},i[r].l=1*new Date();a=s.createElement(o),m=s.getElementsByTagName(o)
[0]; \
a.async=1;a.src=g;m.parentNode.insertBefore(a,m)})(window,document,'script', \
'https://www.google-analytics.com/analytics.js','ga');

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

</script>
```

Copy this script (from the Google Analytics page) and paste it in TODO 3 in **index.html** and **pages/other.html**. Save the scripts and refresh the **app** page (you can close the **page-push-notification.html** page that was opened from the notification click).

Now return to the Google Analytics site. Examine the real time data by selecting **Real-Time** and then **Overview**:



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



Note: If you don't see this, refresh the **app** page and check again.
The **Active Page** indicates which page is being viewed. Back in the app, click **Other page** to navigate to the other page. Then return to the Google Analytics site and check **Active Page** again. It should now show **app/pages/other.html** (this might take a few seconds).

Explanation

When a page loads, the tracking snippet script is executed. The Immediately Invoked Function Expression (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.

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." Because the tracker was created with your tracking ID, this data is sent to your account and property.

Real-time mode in the Google Analytics dashboard shows the hit received from this script execution, along with the page (**Active Page**) that it was executed on.

You can read this documentation to learn more about how **analytics.js** works.

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
- The time spent on each page and the order in which the pages are visited
- Which internal links are clicked (based on the URL of the next pageview)

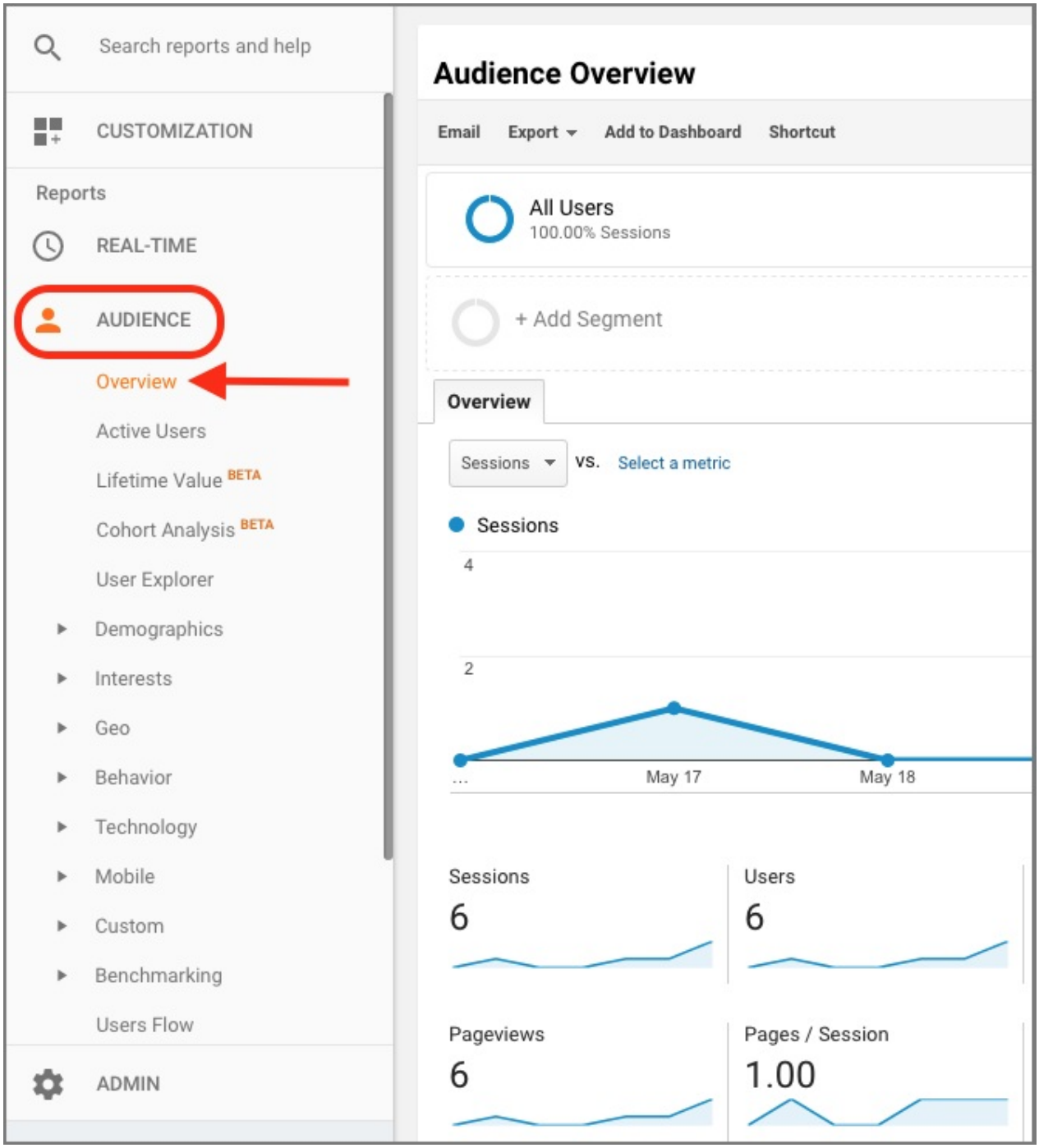
For more information

- The tracking snippet
- Tracker objects
- Creating trackers
- The create command
- The send command
- Hits
- The data sent in a hit
- How analytics.js works

4. View 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 by selecting **Audience** and then **Overview**.

Note: Data for our app is not available yet. It takes some time to process the data, typically 24-48 hours.
Here you can see general information such as pageview records, bounce rate, ratio of new and returning visitors, and other statistics.



You can also see 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

5. Use Debug Mode

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

TODO: Replace **analytics.js** in the tracking snippet (in **index.html** and **pages/other.html**) with **analytics_debug.js**.

Note: Don't use **analytics_debug.js** in production. It is much larger than **analytics.js**.
Save the scripts and refresh the page. You should see the browser console logging details of the "create" and "send" commands.

Note: There is also a Chrome debugger extension that can be used alternatively.
Navigate back to **app/index.html** using the **Back** link. Check the console logs again. Note how the location field changes on the data sent by the send command.

For more information

- Chrome debugger extension
- Debugging Google Analytics

6. Add custom events

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

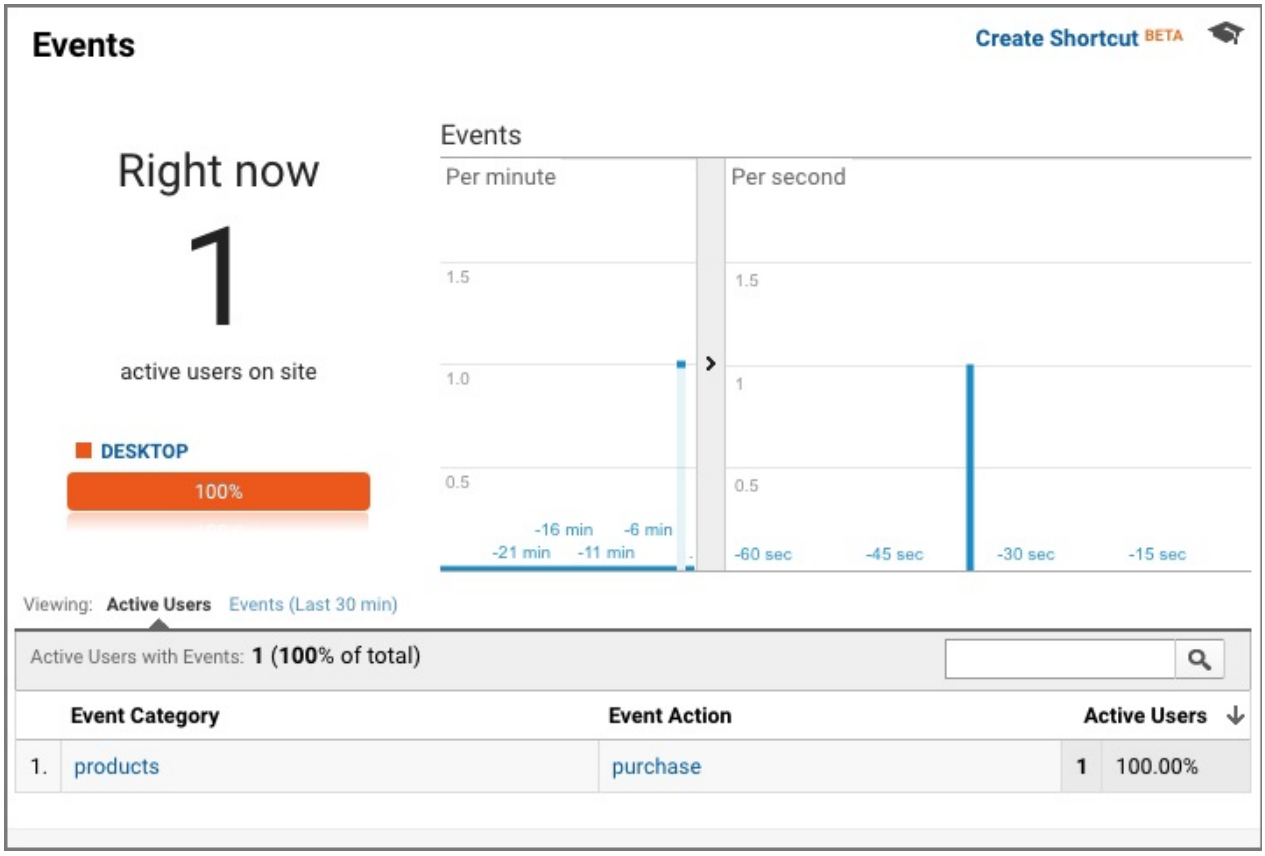
In **main.js**, replace TODO 6 with the following:

main.js

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

Save the script and refresh the page. Click **BUY NOW!!!**. Check the console log, do you see the custom event?

Now return to the **Real-Time** reporting section of the Google Analytics dashboard. Instead of selecting **Overview**, select **Events**. Do you see the custom event? (If not, try clicking **BUY NOW!!!** again.)

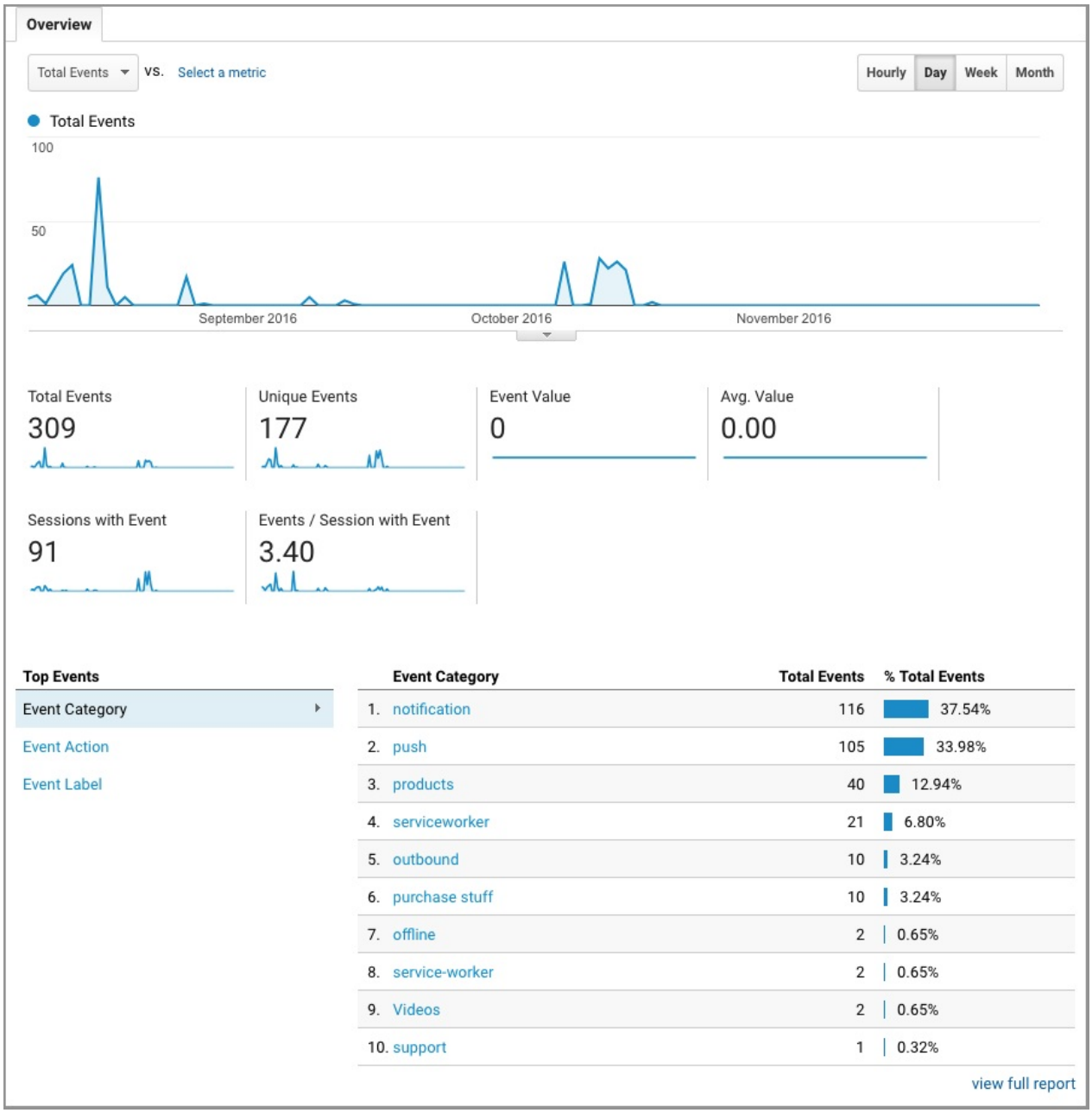


Explanation

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. 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 described in the command queue reference. Hint: Look for the "send" command examples.

You can view past events in the Google Analytics dashboard by selecting **Behavior**, followed by **Events** and then **Overview**. However your account won't yet have any past events to view (because you just created it).



For more information

- Event tracking
- About events
- Command queue reference

7. Showing push notifications

Lets use a custom event to let us know when users subscribe to push notifications.

7.1 Create a project on Firebase

First we need to add push subscribing to our app. To subscribe to the push service in Chrome, you 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. Click the **Settings** (gear) icon next to your project name in the navigation pane, 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 **manifest.json** file with the Sender ID of your Firebase project. The **manifest.json** file should look like this:

manifest.json

```
{
  "name": "Google Analytics codelab",
  "gcm_sender_id": "YOUR_SENDER_ID"
}
```

Save the file. Refresh the app and click **Subscribe**. The browser console should indicate that you have subscribed to push notifications.

Explanation

Chrome uses Firebase Cloud Messaging (FCM) to route its push messages. All push messages are sent to FCM, and then FCM passes them to the correct client.

Note: FCM has replaced Google Cloud Messaging (GCM). Some of the code to push messages to Chrome still contains references to GCM. These references are correct and work for both GCM and FCM.

7.2 Add custom analytics

Now we can add custom analytics events for push subscriptions.

Replace TODO 7.2a with the following code

main.js

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

Replace TODO 7.2b with the following code

main.js

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

Save the script and refresh the app. Now test the subscribe and unsubscribe buttons. Confirm that you see the custom events logged in the browser console, and that they are also shown on the Google Analytics dashboard.

Note that this time we used the alternative send command signature, which is more concise.

Optional: Add analytics hits for the `catch` blocks of the `subscribe` and `unsubscribe` functions. In other words, add analytics code to record when users have errors subscribing or unsubscribing. Then manually block notifications in the app by clicking the icon next to the URL and revoking permission for notifications. Refresh the page and test subscribing, you should see an event fired for the subscription error logged in the console (and in the real-time section of the Google Analytics dashboard). Remember to restore notification permissions when you are done.

Explanation

We have added Google Analytics send commands inside our push subscription code. This lets us track how often users are subscribing and unsubscribing to our push notifications, and if they are experiencing errors in the process.

8. Using analytics in the service worker

The service worker does not have access to the analytics command queue, `ga`, because the command queue is in the main thread (not the service worker thread) and requires the `window` object. We will need to use a separate API to send hits from the service worker.

8.1 Use the Measurement Protocol interface

In **analytics-helper.js**, replace TODO 8.1a with the following code, but use your analytics tracking ID instead of `UA-XXXXXXXX-Y`:

analytics-helper.js

```
// Set this to your tracking ID
var trackingId = 'UA-XXXXXXXX-Y';
```

Replace TODO 8.1b in the same file with the following code:

analytics-helper.js

```
function sendAnalyticsEvent(eventAction, eventCategory) {
  'use strict';

  console.log('Sending analytics event: ' + eventCategory + '/' + eventAction);

  if (!trackingId) {
    console.error('You need your tracking ID in analytics-helper.js');
    console.error('Add this code:\nvar trackingId = \'UA-XXXXXXXX-X\';');
    // We want this to be a safe method, so avoid throwing unless absolutely necessary
.
    return Promise.resolve();
  }

  if (!eventAction && !eventCategory) {
    console.warn('sendAnalyticsEvent() called with no eventAction or ' +
    'eventCategory.');
    // We want this to be a safe method, so avoid throwing unless absolutely necessary
.
    return Promise.resolve();
  }

  return self.registration.pushManager.getSubscription()
  .then(function(subscription) {
    if (subscription === null) {
```

```
        throw new Error('No subscription currently available.');
      }

      // Create hit data
      var payloadData = {
        // Version Number
        v: 1,
        // Client ID
        cid: subscription.endpoint,
        // Tracking ID
        tid: trackingId,
        // Hit Type
        t: 'event',
        // Event Category
        ec: eventCategory,
        // Event Action
        ea: eventAction,
        // Event Label
        el: 'serviceworker'
      };

      // Format hit data into URI
      var payloadString = Object.keys(payloadData)
      .filter(function(analyticsKey) {
        return payloadData[analyticsKey];
      })
      .map(function(analyticsKey) {
        return analyticsKey + '=' + encodeURIComponent(payloadData[analyticsKey]);
      })
      .join('&');

      // Post to Google Analytics endpoint
      return fetch('https://www.google-analytics.com/collect', {
        method: 'post',
        body: payloadString
      });
    })
    .then(function(response) {
      if (!response.ok) {
        return response.text()
        .then(function(responseText) {
          throw new Error(
            'Bad response from Google Analytics:\n' + response.status
          );
        });
      } else {
        console.log(eventCategory + '/' + eventAction +
          'hit sent, check the Analytics dashboard');
      }
    })
    .catch(function(err) {
      console.warn('Unable to send the analytics event', err);
    });
```

```
}
```

Save the script.

Explanation

Because the service worker does not have access to the analytics command queue, `ga`, we need to use the Google Analytics Measurement Protocol interface. This interface lets us make HTTP requests to send hits, regardless of the execution context.

We start by creating a variable with your tracking ID. This will be used to ensure that hits are sent to your account and property, just like in the analytics snippet.

The `sendAnalyticsEvent` helper function starts by checking that the tracking ID is set and that the function is being called with the correct parameters. After checking that the client is subscribed to push, the hit data is created in the `payloadData` variable:

analytics-helper.js

```
var payloadData = {
  // Version Number
  v: 1,
  // Client ID
  cid: subscription.endpoint,
  // Tracking ID
  tid: trackingId,
  // Hit Type
  t: 'event',
  // Event Category
  ec: eventCategory,
  // Event Action
  ea: eventAction,
  // Event Label
  el: 'serviceworker'
};
```

The **version number**, **client ID**, **tracking ID**, and **hit type** parameters are required by the API. The **event category**, **event action**, and **event label** are the same parameters that we have been using with the command queue interface.

Next, the hit data is formatted into a URI with the following code:

analytics-helper.js

```
var payloadString = Object.keys(payloadData)
.filter(function(analyticsKey) {
  return payloadData[analyticsKey];
})
.map(function(analyticsKey) {
  return analyticsKey + '=' + encodeURIComponent(payloadData[analyticsKey]);
})
.join('&');
```

Finally the data is sent to the API endpoint (**https://www.google-analytics.com/collect**) with the following code:

analytics-helper.js

```
return fetch('https://www.google-analytics.com/collect', {
  method: 'post',
  body: payloadString
});
```

The hit is sent with the Fetch API using a POST request. The body of the request is the hit data.

Note: You can learn more about the Fetch API in the fetch codelab.

For more information

- Measurement Protocol
- Push demo

8.2 Send hits from the service worker

Now that we can use the Measurement Protocol interface to send hits, let's add custom events to the service worker.

Replace TODO 8.2a in **sw.js** with the following code:

sw.js

```
self.importScripts('js/analytics-helper.js');
```

Replace TODO 8.2b in **sw.js** with the following code:

sw.js

```
e.waitUntil(
  sendAnalyticsEvent('close', 'notification')
);
```

Replace TODO 8.2c in **sw.js** with the following code:

sw.js

```
sendAnalyticsEvent('click', 'notification')
```

Replace TODO 8.2d in **sw.js** with the following code:

sw.js

```
sendAnalyticsEvent('received', 'push')
```

Save the script. Refresh the page to install the new service worker. Then close and reopen the app to activate the new service worker (remember to close all tabs and windows running the app).

Now try these experiments and check the console and Google Analytics dashboard for each:

1. Trigger a push notification.
2. Click the notification, and note what happens.
3. Trigger another notification and then close it (with the x in the upper right corner).

Do you see console logs for each event? Do you see events on Google Analytics?

Note: Because these events use the Measurement Protocol interface instead of **analytics_debug.js**, the debug console logs don't appear. You can debug the Measurement Protocol hits with hit validation.

Explanation

We start by using ImportScripts to import the **analytics-helper.js** file with our `sendAnalyticsEvent` helper function. Then we use this function to send custom events at appropriate places (such as when push events are received, or notifications are interacted with). We pass in the `eventAction` and `eventCategory` that we want to associate with the event as parameters.

Note: We have used `event.waitUntil` to wrap all of our asynchronous operations. If unfamiliar, `event.waitUntil` extends the life of an event until the asynchronous actions inside of it have completed. This ensures that the service worker will not be terminated pre-emptively while waiting for an asynchronous action to complete.

For more information

- ImportScripts
- event.waitUntil

9. Use analytics offline

What can you do about sending analytics hits when your users are offline? Analytics data can be stored when users are offline and sent at a later time when they have reconnected. Fortunately, there is an npm package for this.

From the app/ directory, run the following command line command:

```
npm install sw-offline-google-analytics
```

This will import the node module.

In **sw.js** replace TODO 9 with:

sw.js

```
importScripts('path/to/offline-google-analytics-import.js');
goog.offlineGoogleAnalytics.initialize();
```

Where `path/to/offline-google-analytics-import.js` is the path to the **offline-google-analytics-import.js** file in the **node_module** folder.

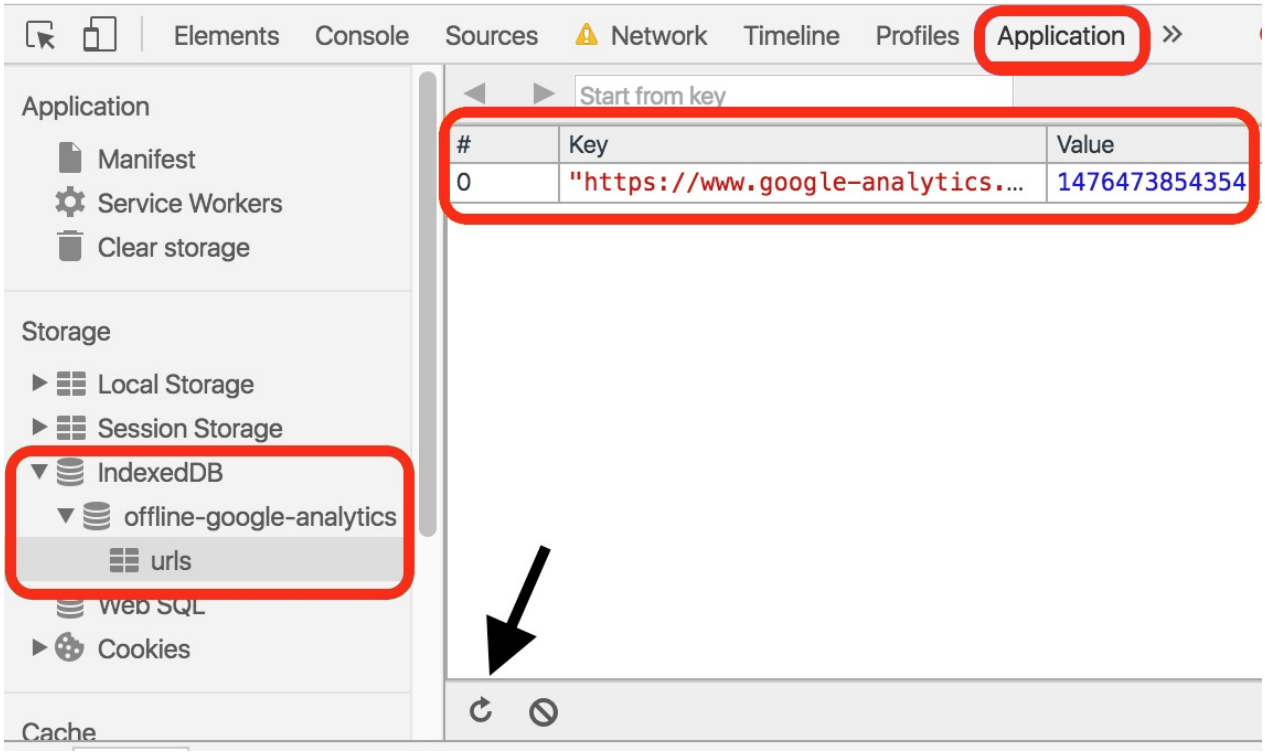
Now save the script. Update the service worker by refreshing the page and closing and reopening the app (remember to close all tabs and windows running the app).

Now simulate offline behavior.

Click **BUY NOW!!!** to fire our first custom analytics event.

You will see an error in the console because we are offline and can't make requests to Google Analytics servers. You can confirm by checking the real-time section of Google Analytics dashboard and noting that the event is not shown.

Now check IndexedDB. Open **offline-google-analytics**. You should see a URL cached. If you are using Chrome (see screenshot below), it is shown in **urls**.You may need to click the refresh icon in the **urls** interface.



Now disable offline mode, and refresh the page. Check **IndexedDB** again, and observe that the URL is no longer cached.

Now check the Google Analytics dashboard. You should see the custom event!

Explanation

Here we import and initialize the **offline-google-analytics-import.js** library. You can check out the documentation for details, but this library 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. The requests are then sent later when connectivity is re-established.

This strategy won't work for hits sent from our service worker because the service worker doesn't listen to fetch events from itself (that could cause some serious problems!). This isn't so important in this case because all the hits that we would want to send from the service worker are tied to online events (like push notifications) anyways.

Note: These events don't use **analytics_debug.js**, so the debug console logs don't appear.
Note: Some users have reported a bug in Chrome that recreates deleted databases on reload.

For more information

- ImportScripts
- Offline Google Analytics
- Google I/O offline example
- IndexedDB

10. Optional: Add hits for notification actions

Add two actions to the push notifications. Send a distinct analytics hit for each action that the user clicks. Remember that you will need to use the Measurement Protocol interface because this code will be in the service worker. Test the actions and make sure the hits are sending.

Note: Notification actions may not be available in Firefox.

11. Optional: Use hitCallback

How can you send analytics hits for an action that takes the user away from your page, such as clicking a link to an external vendor or submitting a form (many browsers stop executing JavaScript once the current page starts unloading, preventing send commands from being executed)?

Research the hitCallback functionality. Use a hitCallback to send an analytics event whenever a user clicks the **Special offers** external link. Make sure to use a timeout so that if the analytics library fails, the user's desired action will still complete!

Note: If the user's browser supports `navigator.sendBeacon` then 'beacon' can be specified as the transport mechanism. This avoids the need for a hitCallback. See the documentation for more info.

For more information

- Sending hits

Solution code

To get a copy of the working code, navigate to the **solution** folder.

Congratulations!

You now know how to integrate Google Analytics into your apps, and how to use analytics with service worker and push notifications.

Resources

- Adding analytics.js to Your Site
- Google Analytics Academy (non-technical)
- Measuring Critical Performance Metrics with Google Analytics code lab
- pageVisibilityTracker plugin (improves pageview and session duration accuracy)

E-Commerce Lab 1: Create a Service Worker

Contents

Overview

1. Get set up

2. Register the service worker

3. Cache the application shell

4. Use the cache-first strategy

5. Delete outdated caches

6. Test it out

Congratulations!

Overview

What you will do

- Register a service worker in your app
- Cache the application shell on service worker install
- Intercept network requests and serve responses from the cache
- Remove unused caches on service worker activation

What you should know

- Basic JavaScript and HTML
- Familiarity with the concept and basic syntax of ES2015 Promises
- Have completed Lab: Scripting the service worker
- Have completed Lab: Fetch API
- Have completed Lab: Caching files with Service Worker

What you will need

- Computer with terminal/shell access
- Connection to the internet
- A browser that supports service workers
- A text editor

1. Get set up

Clone the E-Commerce lab repository with Git using the following command:

```
git clone https://github.com/google-developer-training/pwa-ecommerce-demo.git
```

Note: If you do not use Git, then download the repo from GitHub.
Navigate into the cloned repo:

```
cd pwa-ecommerce-demo
```

If you have a text editor that lets you open a project, then open the **project** folder in the **ecommerce-demo** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **project** folder is where you will build the app.

In a command window at the **project** folder, run the following command to install the project dependencies (open the **package.json** file to see a list of the dependencies):

```
npm install
```

Then run the following:

```
npm run serve
```

This runs the default task in **gulpfile.babel.js** which copies the project files to the appropriate folder and starts a server. Open your browser and navigate to localhost:8080. The app is a mock furniture website, "Modern Furniture Store". Several furniture items should display on the front page.

When the app opens, confirm that a service worker is not registered at local host by checking developer tools. If there is a service worker at localhost, unregister it so it doesn't interfere with the lab.

Note: The e-commerce app is based on Google's Web Starter Kit, which is an "opinionated boilerplate" designed as a starting point for new projects. It allows us to take advantage of

several preconfigured tools that facilitate development, and are optimized both for speed and multiple devices. You can learn more about Web Starter Kit here.
Note: Solution code for this lab can be found in the **solution** folder.

2. Register the service worker

To complete TODO SW-2 in **app/scripts/main.js**, write the code to register the service worker at **service-worker.js**. The code should include a check for whether service worker is supported by the browser. Remember to save the file when you have finished.

3. Cache the application shell

To complete TODO SW-3 in **app/service-worker.js**, write the code to cache the following list of files in the service worker install event. Name the cache `e-commerce-v1`.

```
'/',
'index.html',
'scripts/main.min.js',
'styles/main.css',
'images/products/BarrelChair.jpg',
'images/products/C10.jpg',
'images/products/Cl2.jpg',
'images/products/CP03_blue.jpg',
'images/products/CPC_RECYCLED.jpg',
'images/products/CPFS.jpg',
'images/products/CPO2_red.jpg',
'images/products/CPT.jpg',
'images/products/CS1.jpg',
'images/touch/apple-touch-icon.png',
'images/touch/chrome-touch-icon-192x192.png',
'images/touch/icon-128x128.png',
'images/touch/ms-touch-icon-144x144-precomposed.png',
'images/about-hero-image.jpg',
'images/delete.svg',
'images/footer-background.png',
'images/hamburger.svg',
'images/header-bg.jpg',
'images/logo.png'
```

Save the file.

4. Use the cache-first strategy

To complete TODO SW-4 in **app/service-worker.js**, write the code to respond to fetch requests with the "cache, falling back to network" strategy. First, look for the response in the cache and if it exists, respond with the matching file. If the file does not exist, request the file from the network and cache a clone of the response. Save the file when you have completed this step.

Note: Solution code can be found in the **lab2-add-to-homescreen** folder.

5. Delete outdated caches

To complete TODO SW-5 in **app/service-worker.js**, write the code to delete unused caches in the `activate` event handler. You should create a "whitelist" of caches currently in use that should not be deleted (such as the `e-commerce-v1` cache). Use `caches.keys()` to get a list of the cache names. Then, inside `Promise.all`, map the array containing the cache names to a function that deletes each cache not in the whitelist. Save the file when you have completed this step.

Note: If you get stuck, you can use Lab: Caching files with Service Worker for clues.

6. Test it out

To test the app, close any open instances of the app in your browser and stop the local server (`ctrl+c`).

Run the following in the command line to clean out the old files in the **dist** folder, rebuild it, and serve the app:

```
npm run serve
```

Open the browser and navigate to localhost:8080. Inspect the cache to make sure that the specified files are cached when the service worker is installed. Take the app offline and refresh the page. The app should load normally!

Congratulations!

You have added a service worker to the E-Commerce App. In the sw-precache and sw-toolbox lab, we will generate a service worker in our build process to accomplish the same result with less code.

E-Commerce Lab 2: Add to Homescreen

Contents

Overview

1. Get set up

2. Write the manifest.json file

3. Add to Homescreen elements for Safari on iOS

4. Tile icon for Windows

5. Test it out

Congratulations!

Overview

What you will do

- Integrate the "Add to Homescreen" feature into the e-commerce app

What you should know

- Basic JavaScript and HTML
- Basic JSON

What you will need

- Computer with terminal/shell access
- Connection to the internet
- A browser that supports service workers
- A text editor

1. Get set up

If you have a text editor that lets you open a project, then open the **project** folder in the **ecommerce-demo** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **project** folder is where you will build the app.

If you have completed the previous e-commerce E-Commerce lab, your app is already set up and you can skip to step 2.
If you did not complete lab 1, copy the contents of the **lab2-add-to-homescreen** folder and overwrite the contents of the **project** directory. Then run `npm install` in the command line at the **project** directory.

At the project directory, run `npm run serve` to build the application in **dist**. You must rebuild the application each time you want to test changes to your code. Open your browser and navigate to localhost:8080.

Note: The e-commerce app is based on Google's Web Starter Kit, which is an "opinionated boilerplate" designed as a starting point for new projects. It allows us to take advantage of several preconfigured tools that facilitate development, and are optimized both for speed and multiple devices. You can learn more about Web Starter Kit here.
Note: Solution code for this lab can be found in the **solution** folder.

2. Write the manifest.json file

The web app manifest is a JSON file that lets you, the developer, control how your app appears to the user.

Paste the following contents into the **app/manifest.json** file:

manifest.json

```
{
  "name": "E-Commerce Demo",
  "short_name": "Demo",
  "icons": [{
        "src": "images/touch/icon-128x128.png",
        "sizes": "128x128",
        "type": "image/png"
      }, {
        "src": "images/touch/apple-touch-icon.png",
        "sizes": "152x152",
        "type": "image/png"
      }, {
        "src": "images/touch/ms-touch-icon-144x144-precomposed.png",
        "sizes": "144x144",
        "type": "image/png"
      }, {
        "src": "images/touch/chrome-touch-icon-192x192.png",
        "sizes": "192x192",
        "type": "image/png"
      }],
  "start_url": "/index.html?homescreen=1",
  "display": "standalone",
  "background_color": "#3E4EB8",
  "theme_color": "#2F3BA2"
}
```

Save the file.

Note: The **index.html** file already includes a link to the **manifest.json** file in the head.

3. Add to Homescreen elements for Safari on iOS

Replace TODO HS-3 in **app/index.html** with the following code:

index.html

```
<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="E-Commerce Demo">
<link rel="apple-touch-icon" href="images/touch/apple-touch-icon.png">
```

Save the file.

Explanation

See Configuring Web Applications for a full explanation of each of these elements.

4. Add a tile icon for Windows

Replace TODO HS-4 in **app/index.html** with the following code:

index.html

```
<meta name="msapplication-TileImage" content="images/touch/ms-touch-icon-144x144-preco
mposed.png">
<meta name="msapplication-TileColor" content="#2F3BA2">
```

Save the file.

Explanation

See the Pinned site metadata reference for an explanation of these elements.

5. Test it out

To test the app, close any open instances of the app running in your browser and stop the local server (`ctrl+c`) running in your terminal window.

Run the following in the command line to clean out the old files in the **dist** folder, rebuild it, and serve the app:

```
npm run serve
```

Open your browser to localhost:8080. Unregister the service worker and refresh the page.

If you have Chrome installed, you can test the Add to homescreen functionality from the browser. Open DevTools and inspect the manifest by going to **Application**. Then click **Manifest** in the navigation bar. Click **Add to homescreen**. You should see an "add this site to your shelf" message below the URL bar. This is the desktop equivalent of mobile's add to homescreen feature. If you can successfully trigger this prompt on desktop, then you can be assured that mobile users can add your app to their devices. Click **Add** to install the app on your device.

Congratulations!

You've integrated the Add to homescreen functionality to the E-Commerce App.

E-Commerce Lab 3: Payment Request API

Contents

Overview

1. Get set up

2. Create a PaymentRequest

3. Add payment methods

4. Add payment details

5. Complete the PaymentRequest

6. Allow shipping options

7. Add event handlers

8. Add payment options

9. Test it out

Congratulations!

Overview

What you will do

- Integrate the Payment Request API in the e-commerce app

What you should know

- Basic JavaScript and HTML
- Familiarity with the concept and basic syntax of ES2015 Promises

What you will need

- USB cable to connect an Android device to your computer
- Computer with terminal/shell access
- Connection to the internet

- Chrome for Android version 56 or higher
- A text editor

1. Get set up

If you have a text editor that lets you open a project, then open the **project** folder in the **pwa-ecommerce-demo** folder. This will make it easier to stay organized. Otherwise, open the folder in your computer's file system. The **project** folder is where you will build the app.

If you have completed the E-Commerce App labs up to this point, your app is already set up and you can skip to step 2.
If you did not complete the previous labs, copy the contents of the **lab3-payments** folder and overwrite the contents of the **project** directory. Then run `npm install` in the command line at the **project** directory.

At the project directory, run `npm run serve` to build the application in **dist**. Open your browser and navigate to `localhost:8080` to see the initial state of the app.

Note: The e-commerce app is based on Google's Web Starter Kit, which is an "opinionated boilerplate" designed as a starting point for new projects. It allows us to take advantage of several preconfigured tools that facilitate development, and are optimized both for speed and multiple devices. You can learn more about Web Starter Kit here.
Note: Solution code for this lab can be found in the **solution** folder.
From here, you will be implementing the Payment Request API.

The Payment Request API is not yet supported on desktop as of Chrome 58, so you will need an Android device with Chrome installed to test the code. Follow the instructions in the Access Local Servers article to set up port forwarding on your Android device. This lets you host the e-commerce app on your phone.

2. Create a PaymentRequest

2.1 Detect feature availability

First, let's add add a feature detection for the Payment Request API. And if it's available, let a user process payment with it.

Replace "TODO PAY-2.1" in **app/scripts/modules/app.js** with the following code and remove the dummy conditional of `if (false) {` to add PaymentRequest feature detection:

app.js

```
if (window.PaymentRequest) {
```

Explanation

The feature detection is as simple as examining if `window.PaymentRequest` returns `undefined` or not.

2.2 Create a PaymentRequest

Create a Payment Request object using the PaymentRequest constructor.

Replace TODO PAY-2.2 in **app/scripts/modules/payment-api.js** with the following code to create a new `PaymentRequest` object:

payment-api.js

```
let request = new window.PaymentRequest(supportedInstruments, details, paymentOptions)
;
```

Save the file.

Explanation

The constructor takes three parameters.

supportedInstruments: The first argument is a required set of data about supported payment methods. This can include basic credit cards as well as payment processors like Android Pay. We'll use only basic credit cards in this codelab.

details: The second argument is required information about the transaction. This must include the information to display the total to the user (i.e., a label, currency, and value amount), but it can also include a breakdown of items in the transaction.

paymentOptions: The third argument is an optional parameter for things like shipping. This allows you to require additional information from the user, like payer name, phone, email, and shipping information.

Try it out

You should now be able to try the Payment Request API. If you are not running your server, `npm run serve` and try it using your Android device. Follow the instructions in the Access Local Servers article to set up port forwarding on your Android device.

```
$ npm run serve
```

The `PaymentRequest` UI displays when you click **Checkout**.

Note: The information you enter here won't be posted anywhere other than your local server, but you should use fake information. However, since credit card information requires validation, you can use following fake number so it can accept a random CVC: `4242 4242 4242 4242`
Be aware, this is just the first step and there is more work to be done for the API to complete successfully. Let's continue.

3. Add payment methods

At this point, most of the arguments are empty arrays or objects. Let's configure the payment method (`supportedInstruments`) with proper values.

Replace the JSON block below TODO PAY-3 in **app/scripts/modules/payment-api.js** with this:

payment-api.js

```
{
  supportedMethods: ['basic-card'],
  data: {
    supportedNetworks: ['visa', 'mastercard', 'amex',
      'jcb', 'diners', 'discover', 'mir', 'unionpay']
  }
}
```

Save the file.

Explanation

The first argument of the `PaymentRequest` constructor takes a list of supported payment methods as JSON objects.

`supportedMethods` takes a list of supported method names as an array. Supported methods can be `basic-card` or a URL representing a payment app. These are defined in the Payment Method Identifiers specification.

In the case of `basic-card`, `supportedNetworks` under `data` takes a list of supported credit card brands as defined at Card Network Identifiers Approved for use with Payment Request API. This will filter and show only the credit cards available for the user in the Payment Request UI.

4. Add payment details

Next, let's provide information about items a user is trying to purchase.

4.1 Define the details object

The second argument of the `PaymentRequest` constructor takes details about the purchase. It takes a list of items to display and the total price information.

This portion is already implemented in the `buildPaymentDetails()` function in **app/scripts/modules/payment-api.js**. You don't have to do anything at this time, but see what's happening here.

payment-api.js

```
let details = {
  displayItems: displayItems,
  total: {
    label: 'Total due',
    amount: {currency: 'USD', value: String(total)}
  }
  // TODO PAY-6.2 - allow shipping options
};
return details;
```

Save the file.

Explanation

A required `total` parameter consists of a label, currency and total amount to be charged.

An optional `displayItems` parameter indicates how the final amount was calculated.

The `displayItems` parameter is not intended to be a line-item list, but is rather a summary of the order's major components: subtotal, discounts, tax, shipping costs, etc. Let's define it in the next section.

4.2 Define the display items

The `displayItems` variable should be defined based on items added to the cart.

Replace "TODO PAY-4.2" in **app/scripts/modules/payment-api.js** with the following code and remove the existing `let displayItems = [];` :

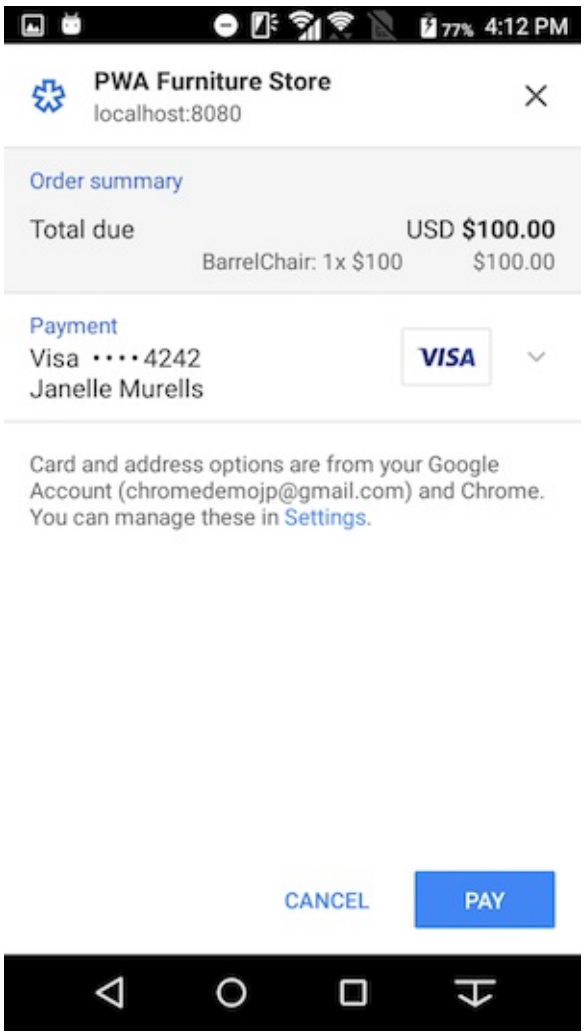
payment-api.js

```
let displayItems = cart.cart.map(item => {
  return {
    label: <code>${item.sku}: ${item.quantity}x $${item.price}</code>,
    amount: {currency: 'USD', value: String(item.total)}
  };
});
```

Save the file.

Explanation

The payment UI should look like this. Try expanding "Order summary":



Notice that the display items are present in the "Order summary" row. We gave each item a `label` and `amount`. `label` is a display label containing information about the item. `amount` is an object that constructs price information for the item.

5. Complete the PaymentRequest

You've put the minimum required options to run a Payment Request. Let's allow a user to complete the payment.

Replace "TODO PAY-5" and the existing `return request.show();` in **app/scripts/modules/payment-api.js** with the following code:

payment-api.js

```
return request.show()
  .then(r => {
    // The UI will show a spinner to the user until
    // <code>request.complete()</code> is called.
    response = r;
    let data = r.toJSON();
    console.log(data);
    return data;
  })
  .then(data => {
    return sendToServer(data);
  })
  .then(() => {
    response.complete('success');
    return response;
  })
  .catch(e => {
    if (response) {
      console.error(e);
      response.complete('fail');
    } else if (e.code !== e.ABORT_ERR) {
      console.error(e);
      throw e;
    } else {
      return null;
    }
  });
```

Explanation

The `PaymentRequest` interface is activated by calling its `show()` method. This method invokes a native UI that allows the user to examine the details of the purchase, add or change information, and pay. A `Promise` (indicated by its `then()` method and callback function) that resolves will be returned when the user accepts or rejects the payment request.

Calling `toJSON()` serializes the response object. You can then POST it to a server to process the payment. This portion differs depending on what payment processor / payment gateway you are using.

Once the server returns a response, call `complete()` to tell the user if processing the payment was successful or not by passing it `success` or `fail`.

Try out the app

Awesome! Now you have completed implementing the basic Payment Request API features in your app. If you are not running your server, `npm run serve` and try it using your Android device.

```
$ npm run serve
```

The `PaymentRequest` UI displays when you click **Checkout**.

6. Allow shipping options

So far you've learned how to integrate the Payment Request API when it doesn't involve shipping items. Moving forward you will learn how to collect shipping information and options from the user.

6.1 Define the shipping options

When you want to collect the user's address information in order to ship items, add `requestShipping: true` in the third property of the `PaymentRequest` constructor.

Replace "TODO PAY-6.1" in **app/scripts/modules/payment-api.js** with the following code:

payment-api.js

```
requestShipping: true,
```

You also need to provide list of shipping options.

Replace "TODO PAY-6.2" in **app/scripts/modules/payment-api.js** with the following code:

payment-api.js

```
,
shippingOptions: displayedShippingOptions
```

Luckily `SHIPPING_OPTIONS` is predefined in the **app/scripts/modules/payment-api.js**; you can parse it and construct the `displayShippingOptions` object from it.

Replace TODO PAY-6.3 in **app/scripts/modules/payment-api.js** with the following code:

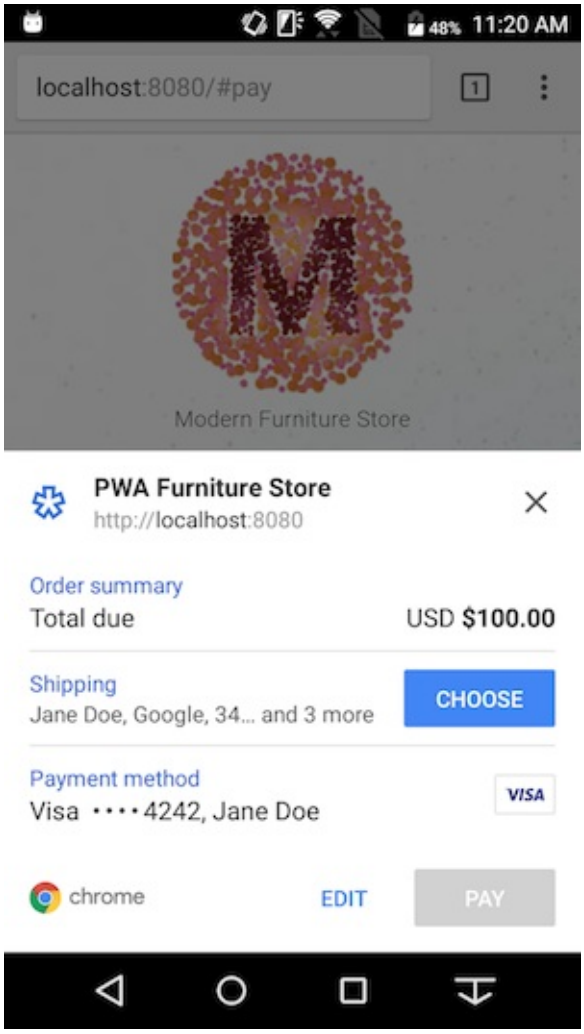
payment-api.js

```
let displayedShippingOptions = [];
if (shippingOptions.length > 0) {
  let selectedOption = shippingOptions.find(option => {
    return option.id === shippingOptionId;
  });
  displayedShippingOptions = shippingOptions.map(option => {
    return {
      id: option.id,
      label: option.label,
      amount: {currency: 'USD', value: String(option.price)},
      selected: option.id === shippingOptionId
    };
  });
  if (selectedOption) total += selectedOption.price;
}
```

Save the file.

Explanation

`id` is a unique identifier of the shipping option item. `label` is a displayed label of the item. `amount` is an object that constructs price information for the item. `selected` is a boolean that indicates if the item is selected.



Notice that these changes add a section to the Payment Request UI, "Shipping". But beware, selecting shipping address will cause UI to freeze and timeout. To resolve this, you will need to handle `shippingaddresschange` event in the next section.

Note: The address information available here is retrieved from the browser's autofill information. Depending on the user's browser status, users will get address information pre-filled without typing any text. They can also add a new entry on the fly.

7. Add event handlers

What if a user specifies a shipping address that's outside of your target countries and not deliverable? How do you charge a user when the user changes a shipping option? The answer is to receive events upon the user's making changes and update with relevant information.

7.1 Add `shippingaddresschange` event listener

When the user changes a shipping address, you will receive the `shippingaddresschange` event.

Replace "TODO PAY-7.1" in **app/scripts/modules/payment-api.js** with the following code:

payment-api.js

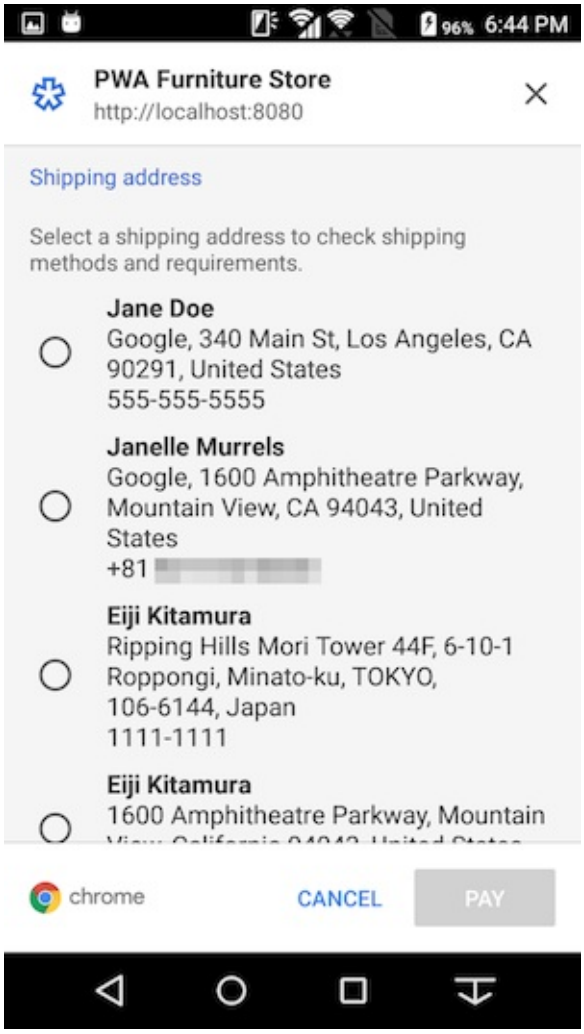
```
// When user selects a shipping address, add shipping options to match
request.addEventListener('shippingaddresschange', e => {
  e.updateWith((_ => {
    // Get the shipping options and select the least expensive
    shippingOptions = this.optionsForCountry(request.shippingAddress.country);
    selectedOption = shippingOptions[0].id;
    let details = this.buildPaymentDetails(cart, shippingOptions, selectedOption);
    return Promise.resolve(details);
  })());
});
```

Explanation

Upon receiving the `shippingaddresschange` event, the `request` object's `shippingAddress` information is updated. By examining it, you can determine if

- The item is deliverable.
- Shipping cost needs to be added/updated.

This code looks into the country of the shipping address and provides free shipping and express shipping inside the US, and provides international shipping otherwise. Checkout `optionsForCountry()` function in **app/scripts/modules/payment-api.js** to see how the evaluation is done.



Note that passing an empty array to `shippingOptions` indicates that shipping is not available for this address. You can display an error message via `shippingOption.error` in that case.

7.2 Add `shippingoptionchange` event listener

When the user changes a shipping option, you will receive the `shippingoptionchange` event.

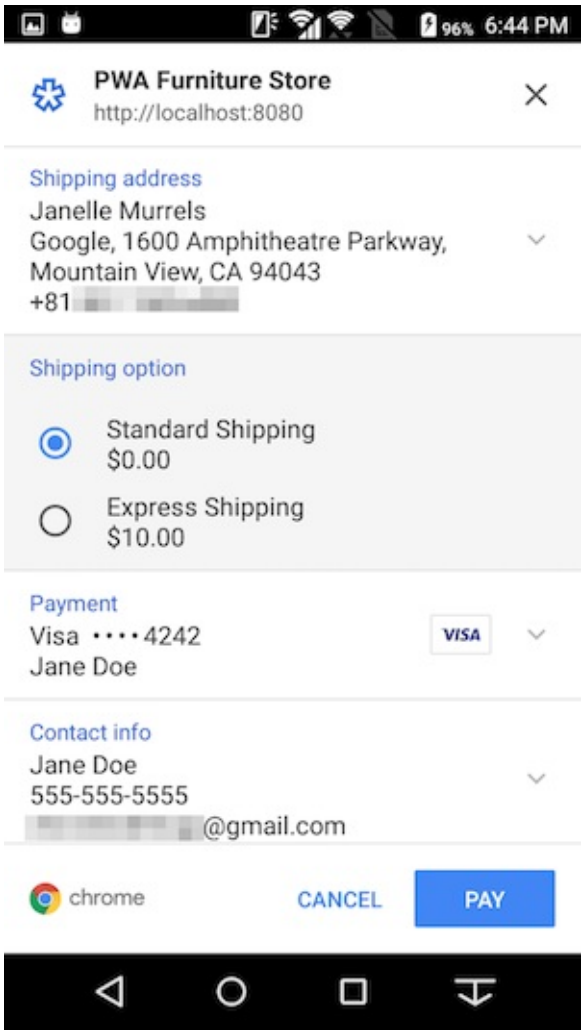
Replace "TODO PAY-7.2" in **app/scripts/modules/payment-api.js** with the following code:

payment-api.js

```
// When user selects a shipping option, update cost, etc. to match
request.addEventListener('shippingoptionchange', e => {
  e.updateWith((_ => {
    selectedOption = request.shippingOption;
    let details = this.buildPaymentDetails(cart, shippingOptions, selectedOption);
    return Promise.resolve(details);
  })());
});
```

Explanation

Upon receiving the `shippingoptionchange` event, the `request` object's `shippingOption` is updated. The `shippingOption` It indicates the `id` of the selected shipping options. The `id` is passed to `buildPaymentsDetails`, which looksLook for the price of the shipping option and updates the display items so that the user knows the total cost is changed. `buildPaymentsDetails alsoAlso` changes the shipping option's `selected` property to `true` to indicate that the user has chosen the item. Checkout `buildPaymentDetails()` function in **app/scripts/modules/payment-api.js** to see how it works.



8. Add payment options

In addition to the shipping address, there are options to collect payer's information such as email address, phone number, and name.

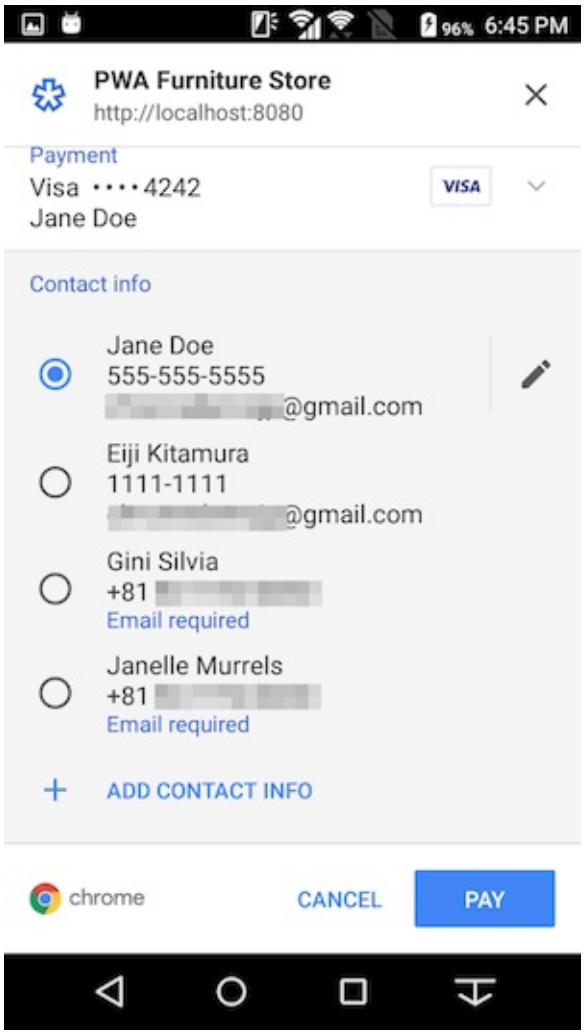
Replace "TODO PAY-8" in **app/scripts/modules/payment-api.js** with the following payment options:

payment-api.js

```
requestPayerEmail: true,
requestPayerPhone: true,
requestPayerName: true
```

Save the file.

Explanation

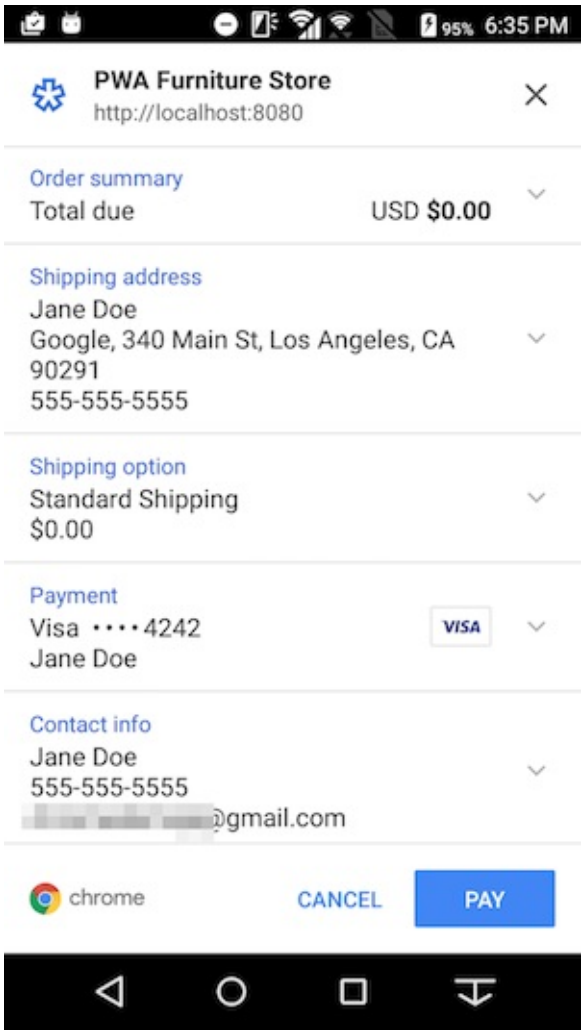


By adding `requestPayerPhone: true`, `requestPayerEmail: true`, `requestPayerName: true` to the third argument of the `PaymentRequest` constructor, you can request the payer's phone number, email address, and name.

9. Test it out

Phew! You have now completed implementing the Payment Request API with shipping option. Let's try it once again by running your server if it's stopped.

```
$ npm run serve
```



Try: add random items to the card, go to checkout, change shipping address and options, and finally make a payment.

Follow the instructions in the Access Local Servers article to set up port forwarding on your Android device. This lets you host the e-commerce app on your phone.

Once you have the app running on your phone, add some items to your cart and go through the checkout process. The `PaymentRequest` UI displays when you click **Checkout**.

The payment information won't go anywhere, but you might be hesitant to use a real credit card number. Use "4242 4242 4242 4242" as a fake one. Other information can be anything.

The service worker is caching resources as you use the app, so be sure to unregister the service worker and run `npm run serve` if you want to test new changes.

Congratulations!

You have added Payment integration to the e-commerce app. Congratulations!

To learn more about the Payment Request API, visit the following links.

Resources

- Bringing Easy and Fast Checkout with Payment Request API
- Payment Request API: an Integration Guide
- Web Payments session video at Chrome Dev Summit 2017

Specs

- Payment Request API
- Payment Handler API

Demos

- https://paymentrequest.show/demo/
- https://googlechrome.github.io/samples/paymentrequest/
- https://woocommerce.paymentrequest.show/

Tools for PWA Developers

Contents

Open Developer Tools

Open the console

Work with the network

Inspect the Manifest

Interact with service workers in the browser

Check notification permissions

Inspect cache storage

Further reading

Open Developer Tools

Chrome

To access **Developer Tools** ("DevTools") in Chrome (https://developer.chrome.com/devtools), open a web page or web app in Google Chrome. Click the **Chrome menu** icon, and then select **More Tools** > **Developer Tools**.

You can also use the keyboard shortcut `Control+Shift+I` on Windows and Linux, or `⌘ + alt + I` on Mac (see the Keyboard and UI Shortcuts Reference). Alternatively, right-click anywhere on the page and select **Inspect**.

On a Mac, you can also select **View** > **Developer** > **Developer Tools** in the Chrome menu bar at the top of the screen.

The **DevTools** window opens in your Chrome browser.

Firefox

To open Developer Tools in Firefox, open a web page or web app in Firefox. Click the Menu icon  in the browser toolbar, and then click **Developer** > **Toggle Tools**.

You can also use the keyboard shortcut `Control+Shift+I` on Windows and Linux, or `⌘ + alt + I` on Mac (see the Keyboard Shortcuts Reference).

On Mac, you can also select **View** > **Web Developer** > **Toggle Tools** in the Firefox menu bar at the top of the screen.

The **Toolbox** window opens in your Firefox browser.

Opera

To launch **Opera Dragonfly**, open a web page or web app in Opera. Use the keyboard shortcut `Ctrl + Shift + I` on Windows and Linux, or `⌘ + alt + I` on Mac. Alternatively, you can target a specific element by right-clicking in the page and selecting "Inspect Element".

On a Mac, you can also select **View** > **Show Developer Menu** in the Opera menu bar at the top of the screen. Then select **Developer** > **Developer Tools**.

The **Dragonfly** window opens in your Opera browser.

Internet Explorer

To open Developer Tools in Internet Explorer, open a web page or web app in Internet Explorer. Press `F12` or click **Developer Tools** from the **Tools** menu.

Safari

To start using **Web Inspector** in Safari, open a web page or web app in Safari. In the menu bar, select **Safari** > **Preferences**. Go to the **Advanced** pane and enable the "Show Develop menu in menu bar" setting. In the menu bar, select **Develop** > **Show Web Inspector**.

You can also use the keyboard shortcut `⌘ + ⌥ + I`.

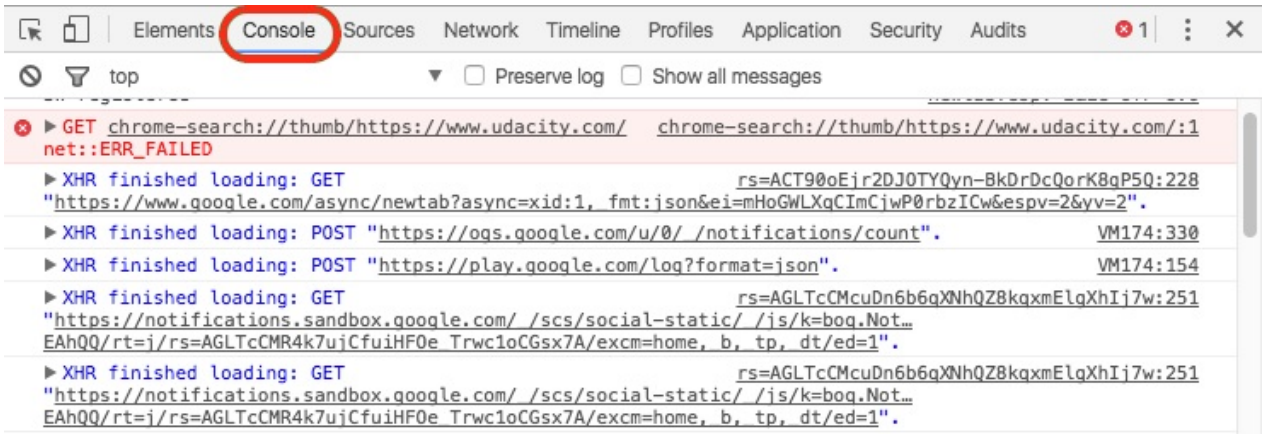
The **Web Inspector** window opens in your Safari browser.

Open the console

Chrome

To open the dedicated **Console** panel, either:

- Press `Ctrl + Shift + J` (Windows / Linux) or `⌘ + ⌥ + J` (Mac).
- Open DevTools and select the **Console** panel.

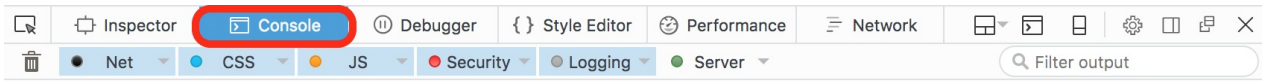


See Using the Console for more information.

Firefox

To open the Web Console, either:

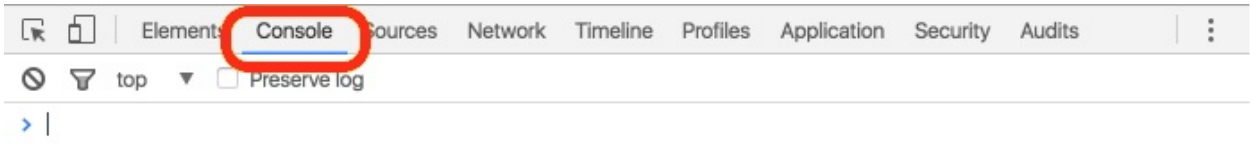
- Press `Ctrl + Shift + K` (Windows/Linux) or `⌘ + ⌥ + K` (Mac).
- From the Firefox menu (or Tools menu if you display the menu bar or are on Mac OS X), select **Developer > Web Console.**
- Open the Toolbox and select the **Console** panel.



See Opening the Web Console for more information.

Opera

Open Dragonfly and select the **Console** panel.



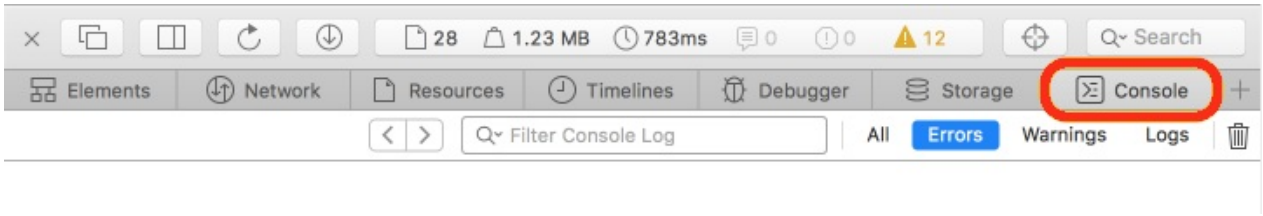
Internet Explorer

Open Developer Tools and select the **Console** panel.

Safari

To open the **Console**, either:

- Enable the Developer menu. From the menu bar, select **Develop** > **Show Error Console**.
- Press `⌘ + ⌥ + C`
- Open the Web Inspector and select the **Console** panel.

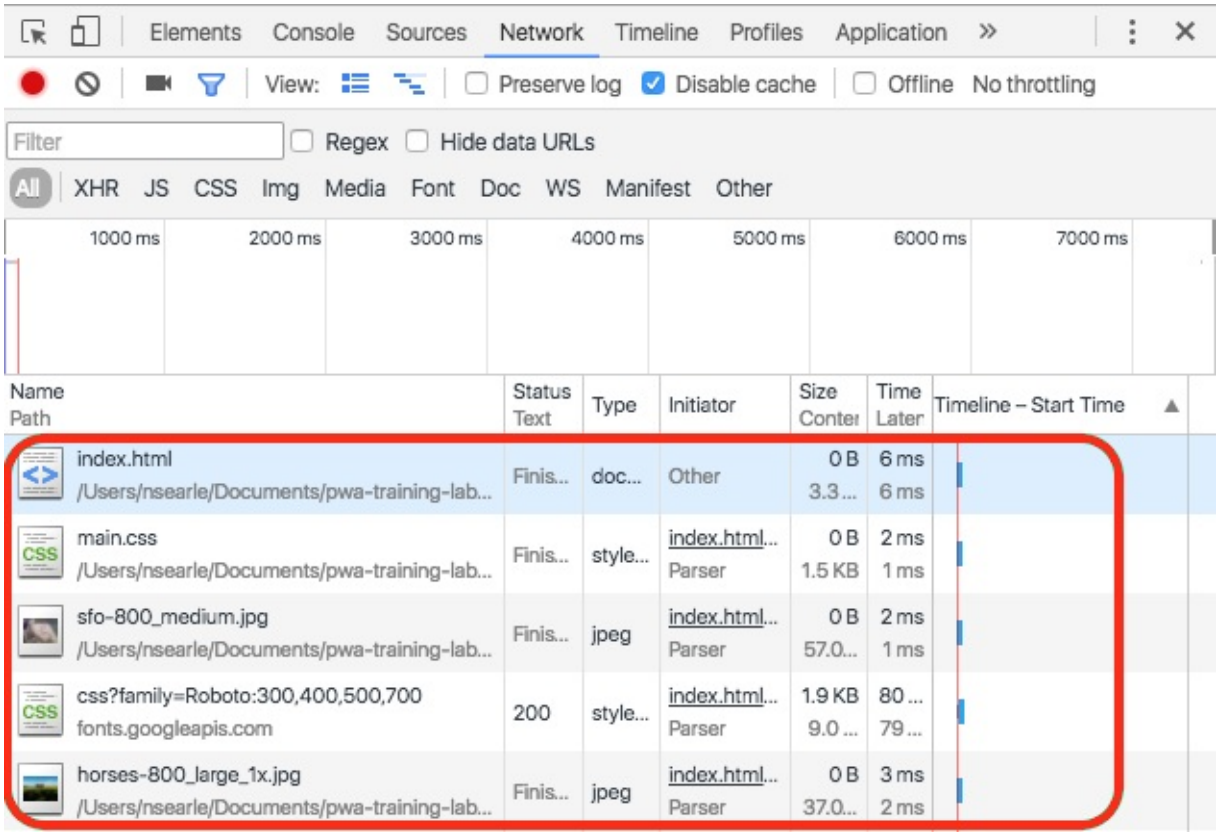


Work with the network

View network requests

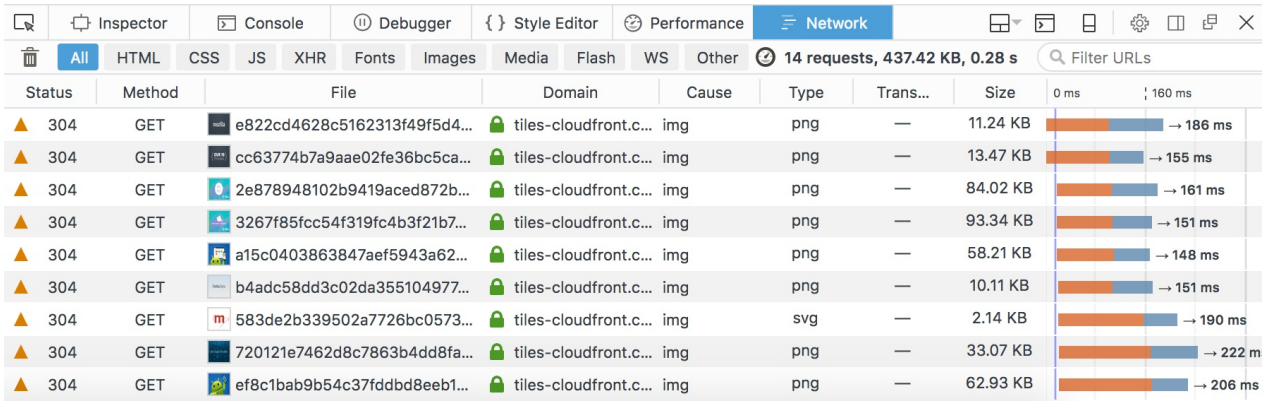
Chrome

Open DevTools and select the **Network** panel. Requests are displayed in the **Network** panel's **Requests Table**. See Measure Resource Loading Times for more information.



Firefox

Open the Toolbox and select the **Network** panel. See Network Monitor for more information.



Opera

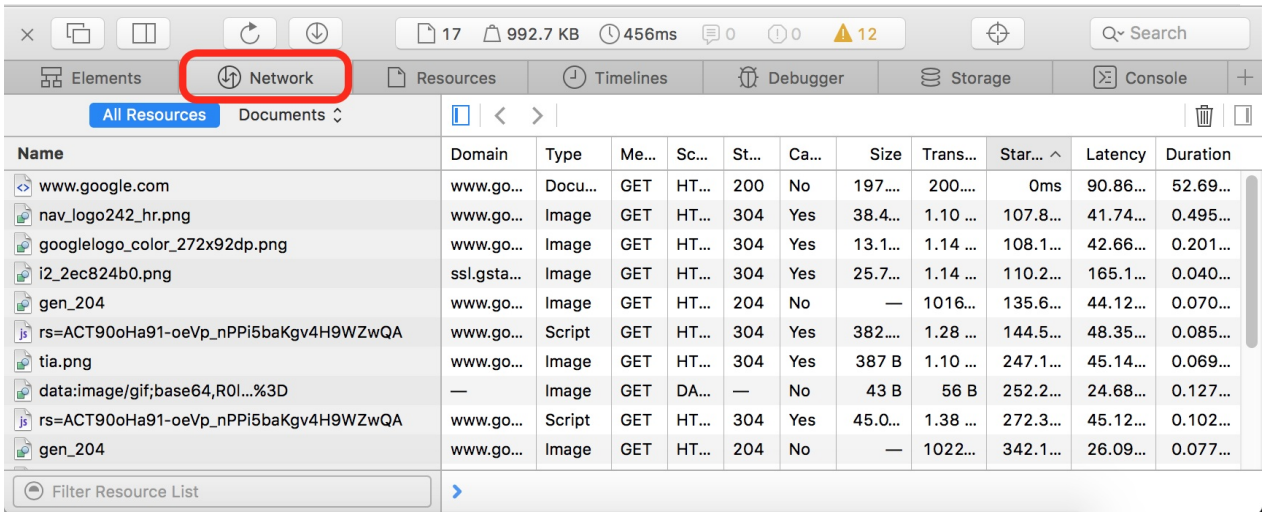
See View Network Requests in Chrome.

Internet Explorer

Open Developer Tools, and then open the **Network** panel. See Network for more information.

Safari

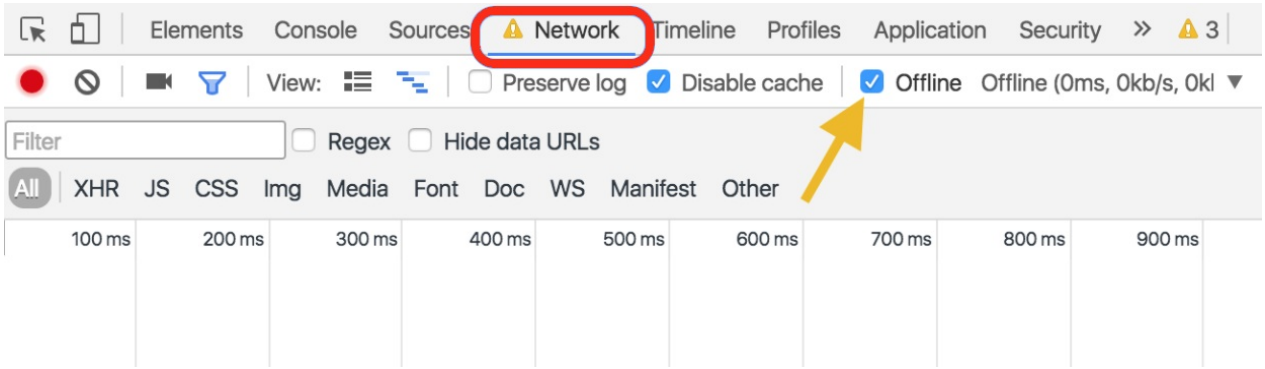
Open the Web Inspector, and then open the **Network** panel.



Simulate offline behavior

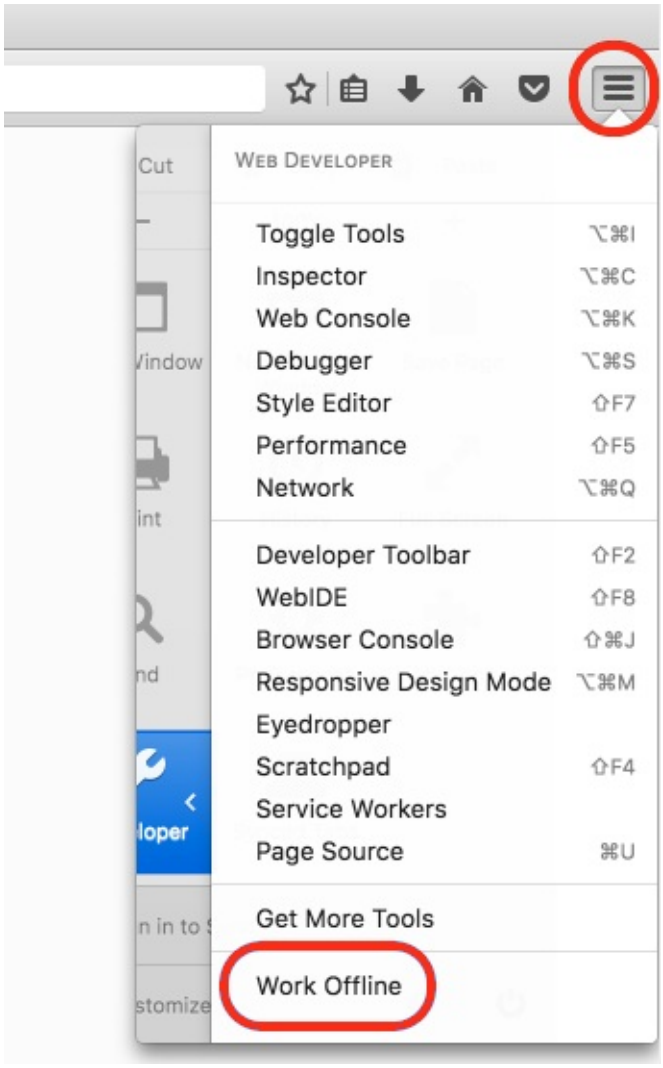
Chrome

Open DevTools and select the **Network** panel. Check the **Offline** checkbox. Check out Optimize Performance Under Varying Network Conditions for more information.

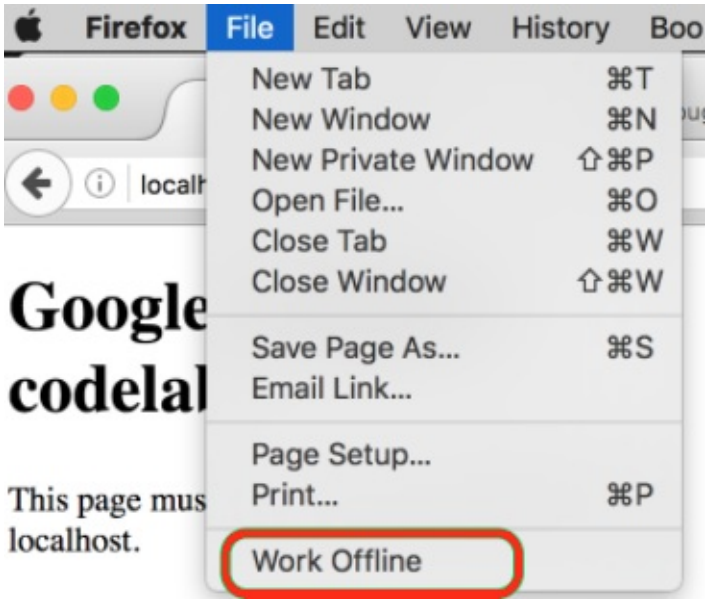


Firefox

Click menu icon ≡ in the browser toolbar. Then click **Developer** > **Work Offline**.



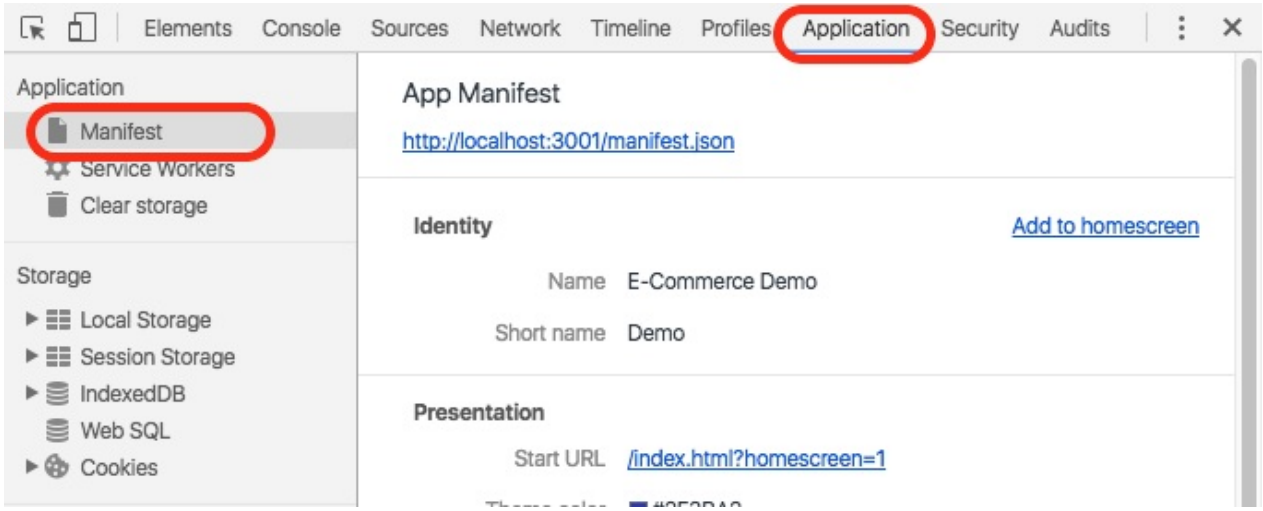
On Mac, you can enable offline mode from the menu bar by clicking **File** > **Work Offline**.



Inspect the manifest

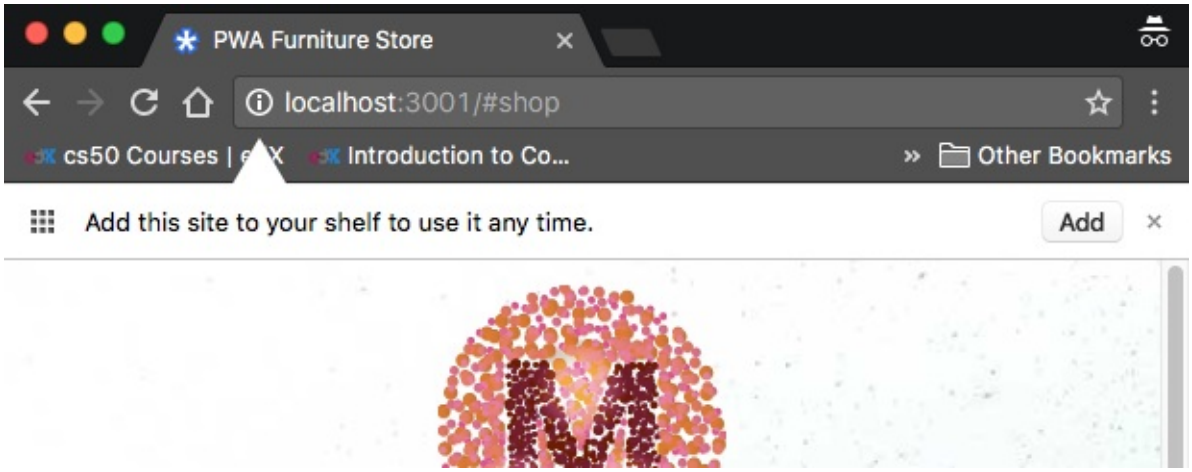
Chrome

Open DevTools in Chrome. Click the **Application** panel, and then click **Manifest** in the navigation bar.



If your app has a **manifest.json** file, the options you have defined will be listed here.

You can test the add to homescreen feature from this pane. Click **Add to homescreen**. You should see an "add this site to your shelf" message.

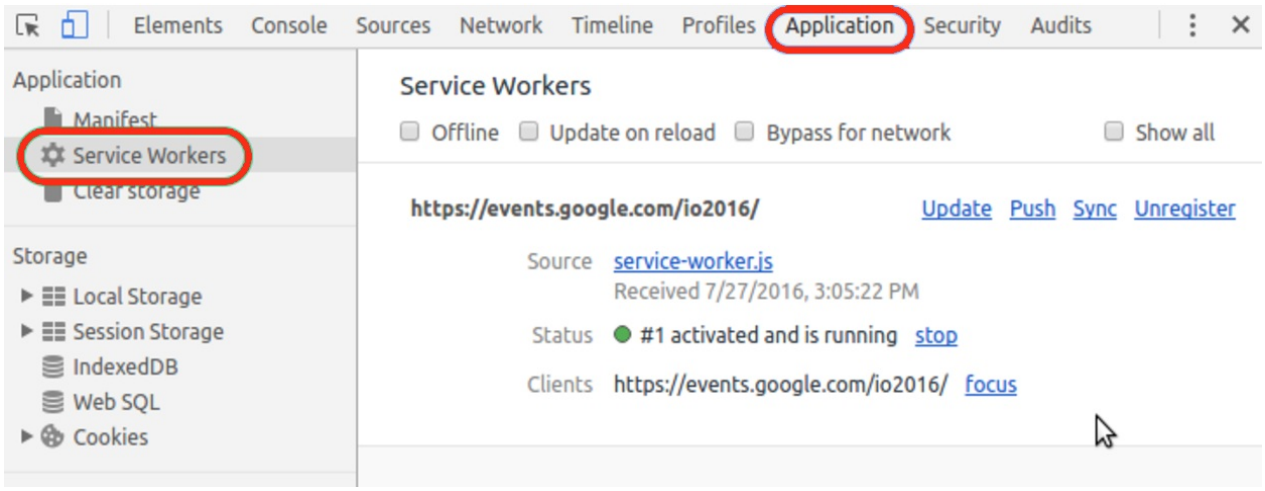


Interact with service workers in the browser

Inspect the service worker

Chrome

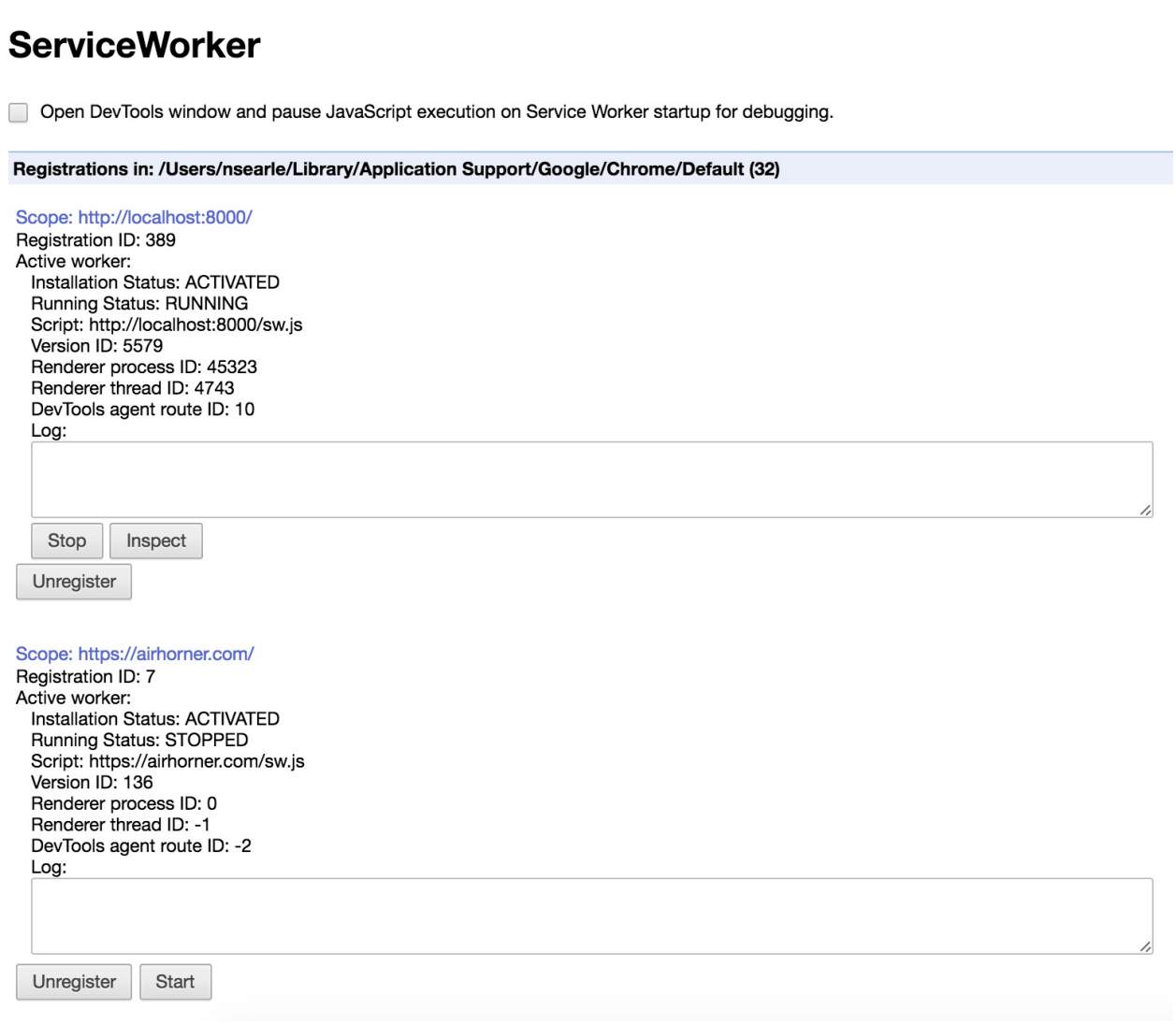
Open DevTools in Chrome. Click the **Application** panel, and then click **Service Workers** in the navigation bar.



If a service worker is installed for the currently open page, you'll see it listed on this pane. For example, in the screenshot above there's a service worker installed for the scope of `https://events.google.com/io2016/`.

chrome://serviceworker-internals/

You can also view a list of all service workers by navigating to **chrome://serviceworker-internals/** in your Chrome browser.



Firefox

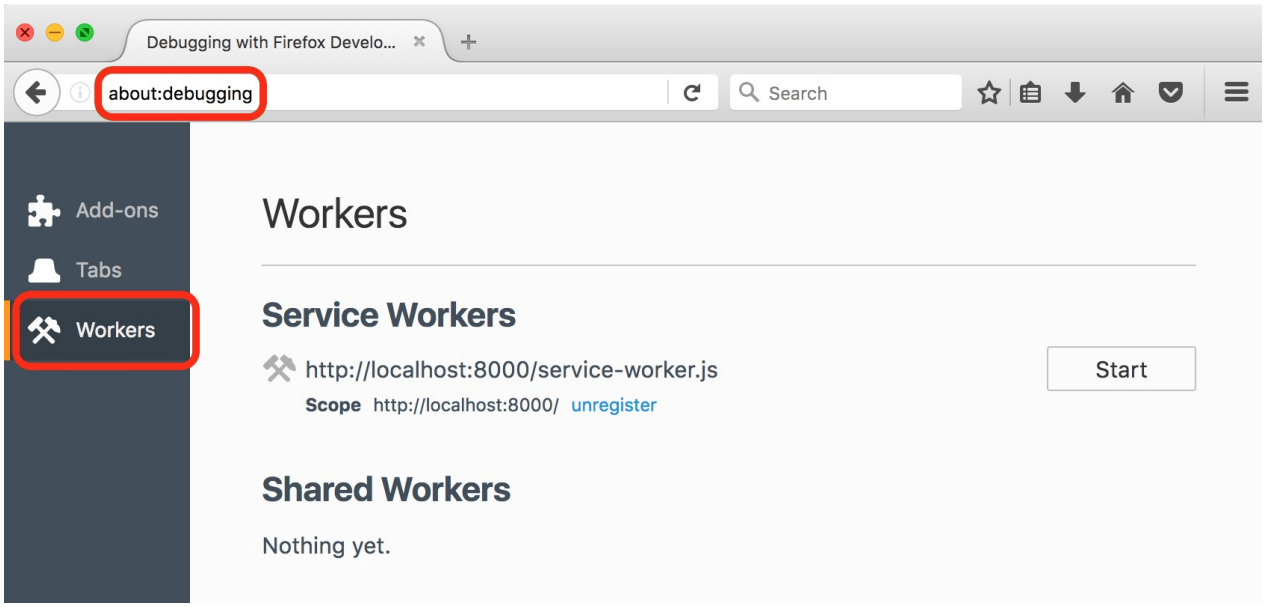
The **about:debugging** page provides an interface for interacting with Service Workers.

There are several different ways to open **about:debugging**:

- On Mac, in the **Tools** > **Web Developer** menu, click **Service Workers**.
- Click the Menu icon ☰ in the browser toolbar.

Then click the Developer icon and select **Service Workers**.

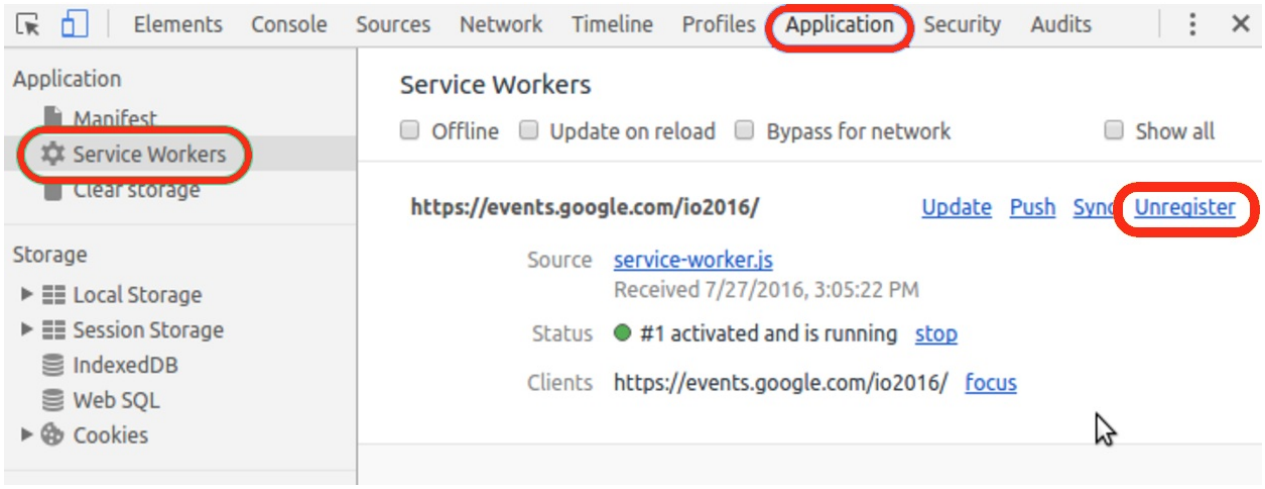
- Enter "about:debugging" in the Firefox URL bar and click **Workers**.



Unregister the service worker

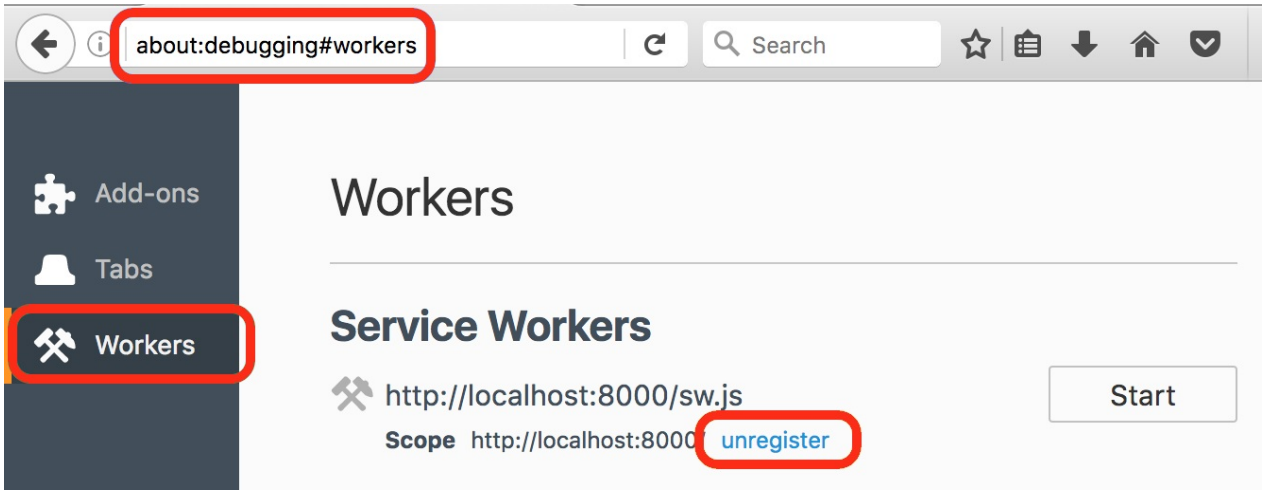
Chrome

Open the Service Workers pane in **DevTools**. Click **Unregister** next to the service worker.



Firefox

Open the Workers page in about:debugging. Click **Unregister** next to the service worker scope.

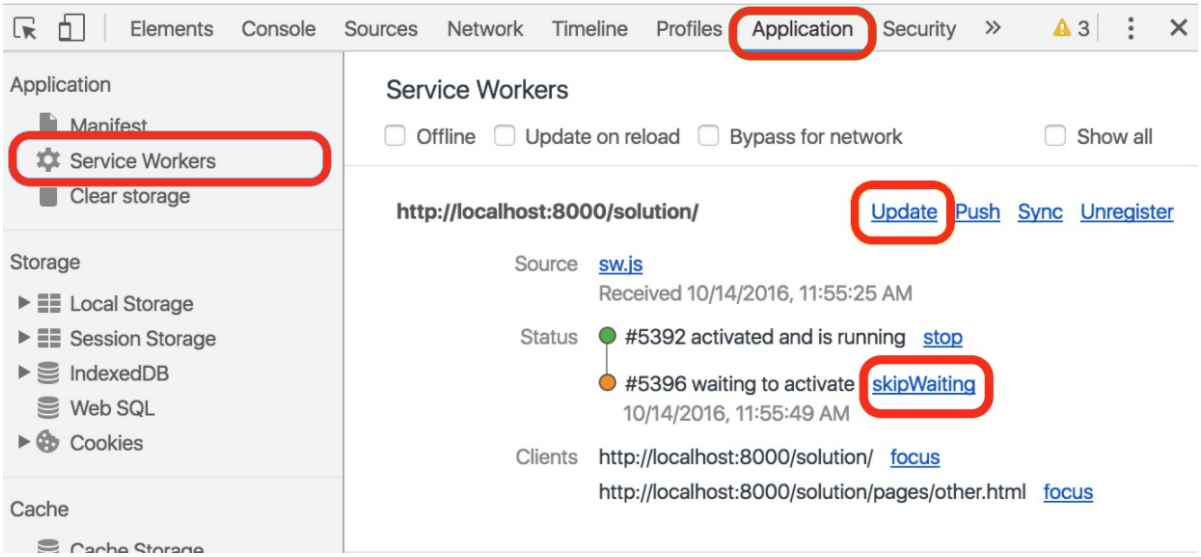


Force update the service worker

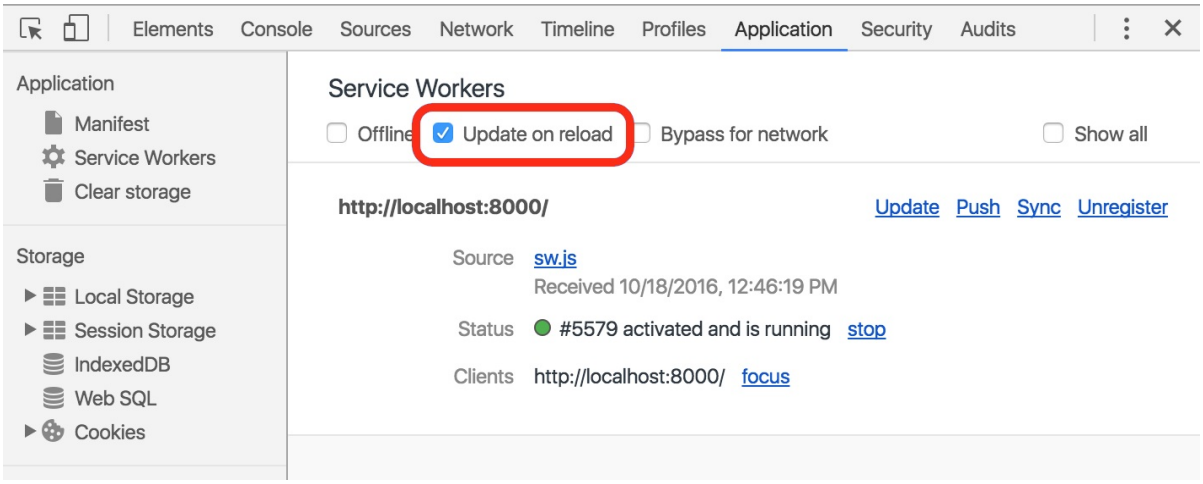
Chrome

There are several ways to force-update the service worker in Chrome:

1. Refresh your app in the browser so the new service worker is recognized. Then hold `Shift` and click the Reload icon .
2. Open the Service Workers pane in **DevTools**. Click **Update**. When the new service worker installs, click **skipWaiting**.



3. To force the service worker to update automatically whenever you reload the page, check **Update on reload**.



4. Unregister the service worker and refresh the app in the browser.

Firefox

To update the service worker in Firefox, close all pages controlled by the service worker and then reopen them. The service worker only updates when there are no pages open in Firefox that are within its scope.

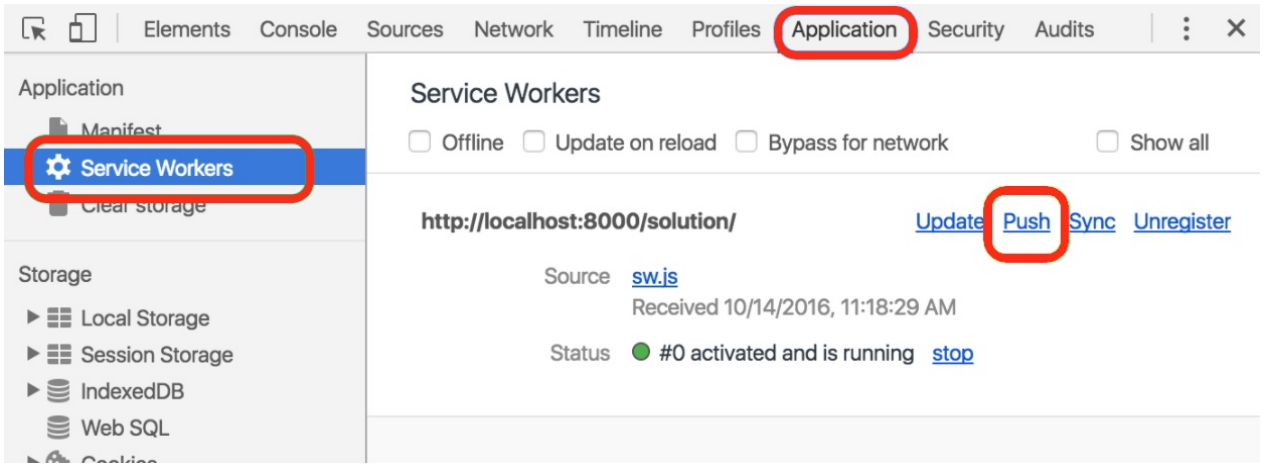
If you want to be absolutely certain (for testing reasons) that the service worker will update, you can unregister the service worker from the **about:debugging** page and refresh your app in the browser. The new service worker installs on page reload.

Note that unregistering the service worker will change the subscription object if you are working with Push Notifications. Be sure to use the new subscription object if you unregister the service worker.

Send simulated push notifications

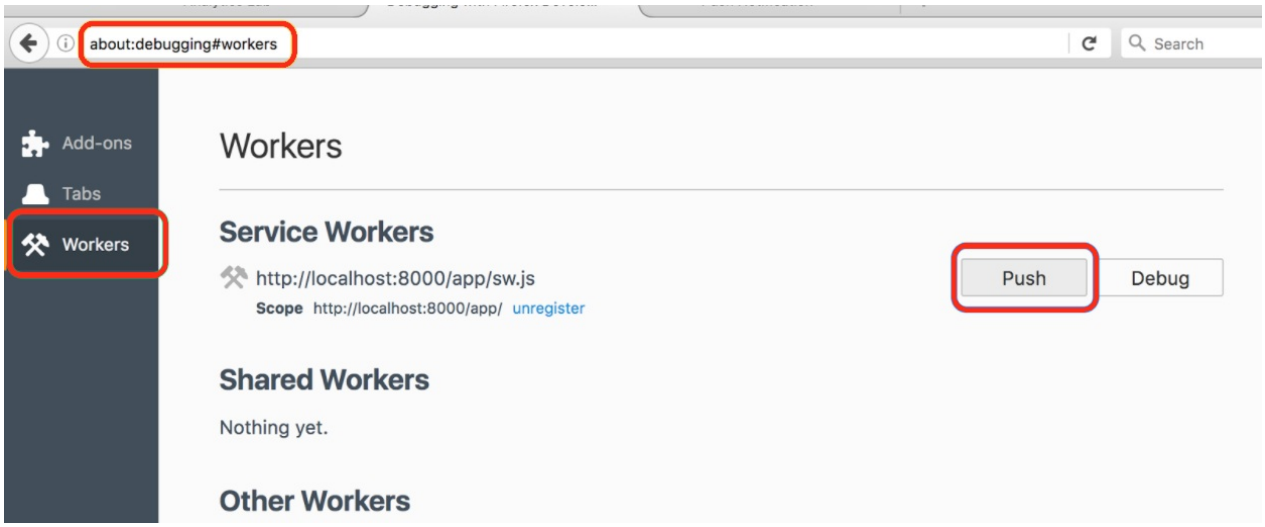
Chrome

Open the Service Workers pane in **DevTools**. Click **Push** to ping the service worker.



Firefox

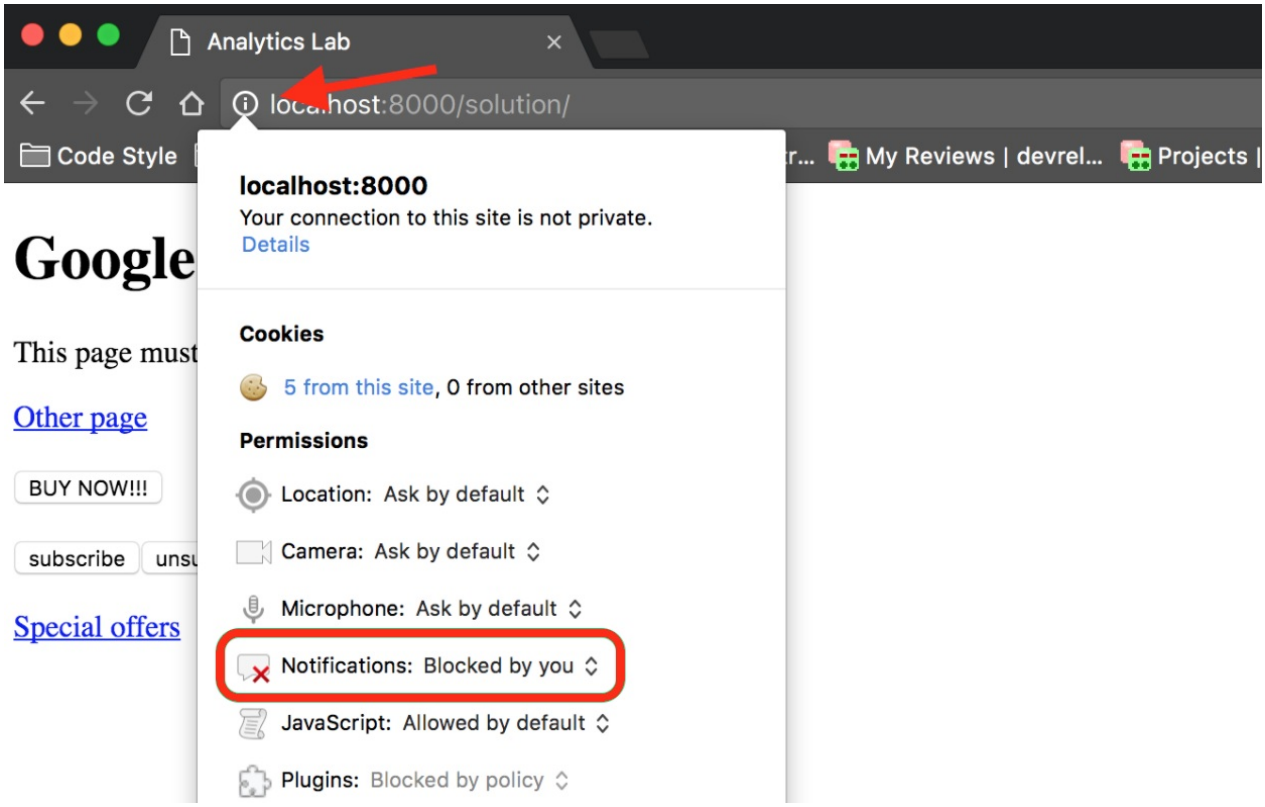
Navigate to **about:debugging** in Firefox and select **Workers**. Click **Push**. If the worker isn't running, you will see **Start** instead of **Push**. Click **Start** to start the service worker, then click **Push**.



Check notification permissions

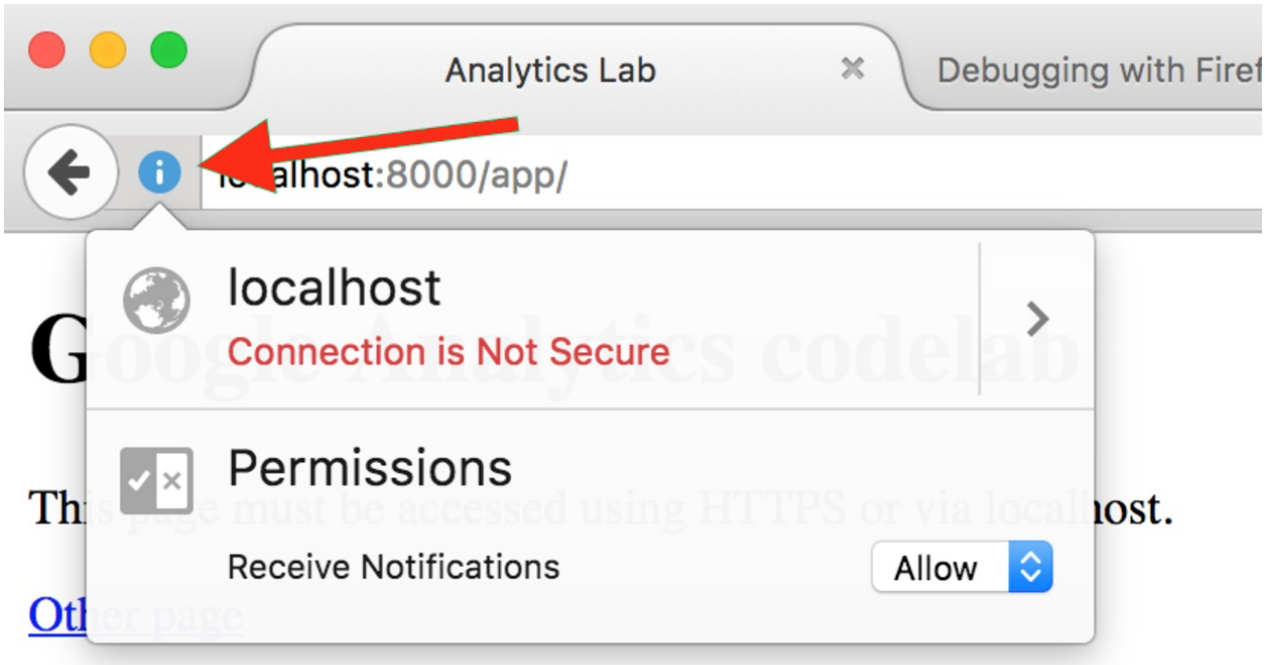
Chrome

Click the Information icon in the URL bar. Use the **Notifications** dropdown menu to set the permission status for **Notifications**.



Firefox

Click the Information icon in the URL bar. Use the **Receive Notifications** dropdown menu to set the permission status for notifications.

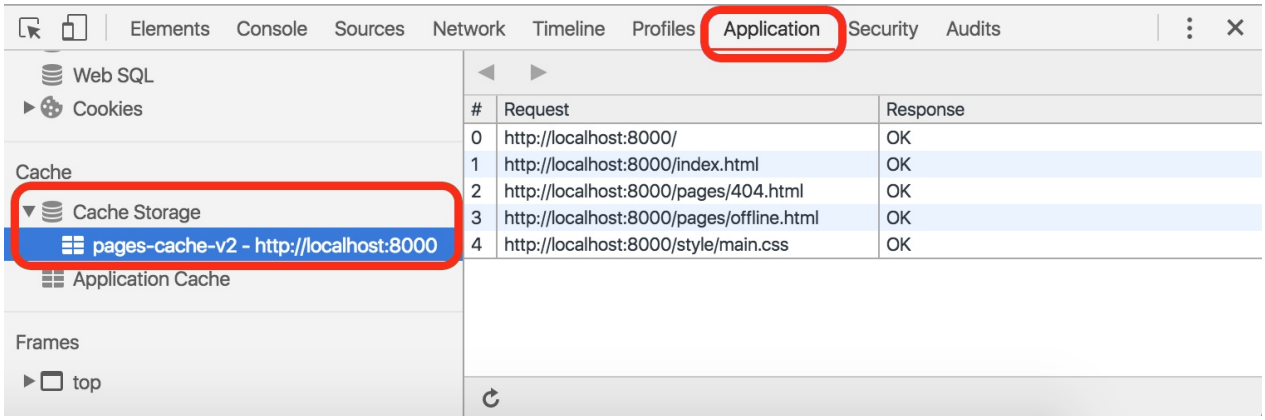


Inspect cache storage

Check the service worker cache

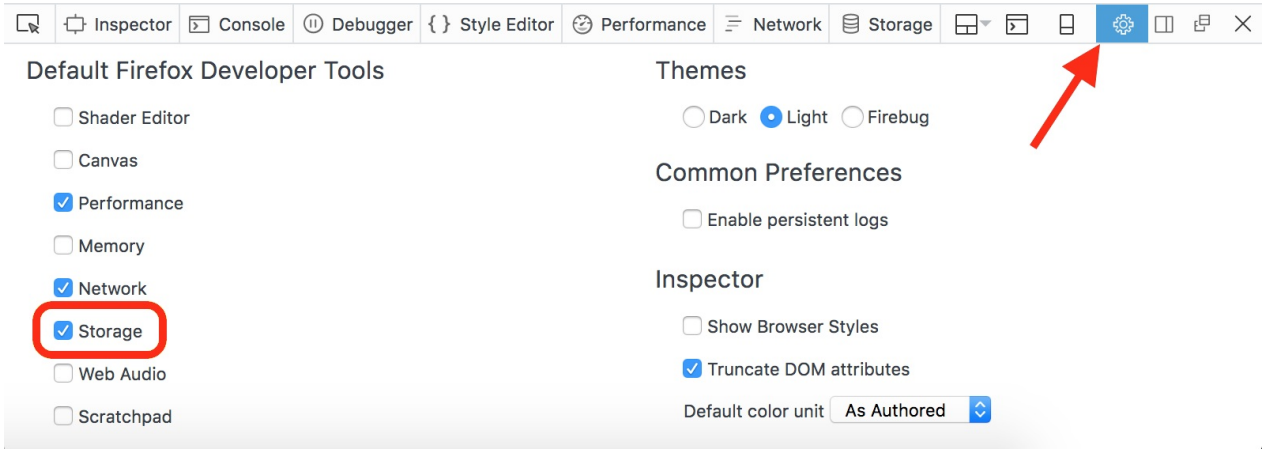
Chrome

Open DevTools and select the **Application** panel. In the navigation bar click **Cache Storage** to see a list of caches. Click a specific cache to see the resources it contains.

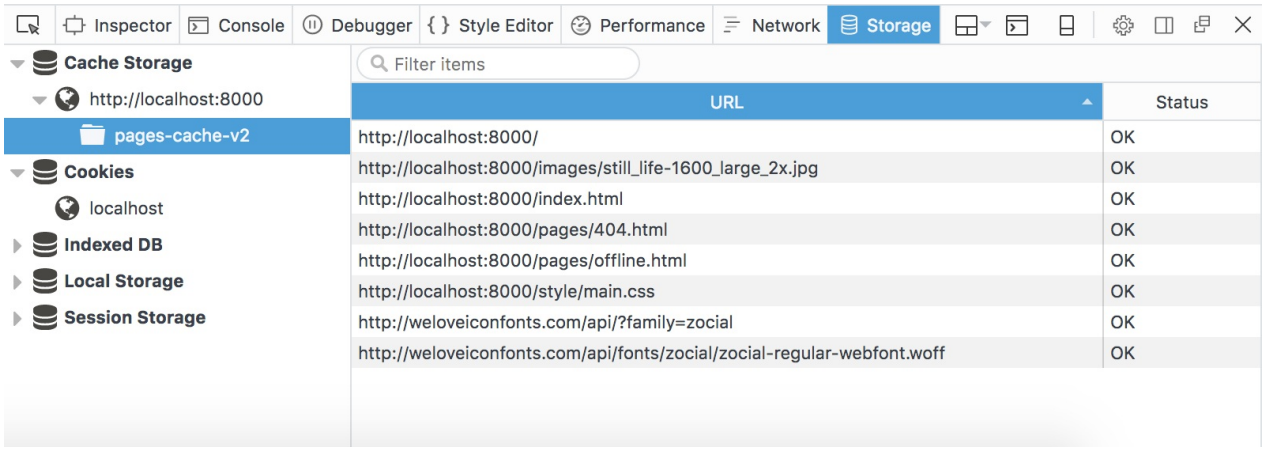


Firefox

Open the Toolbox and click the Settings icon to open **Settings**. Under **Default Firefox Developer Tools**, check **Storage**.



Open the **Storage** panel and expand the **Cache Storage** node. Select a cache to see its contents.

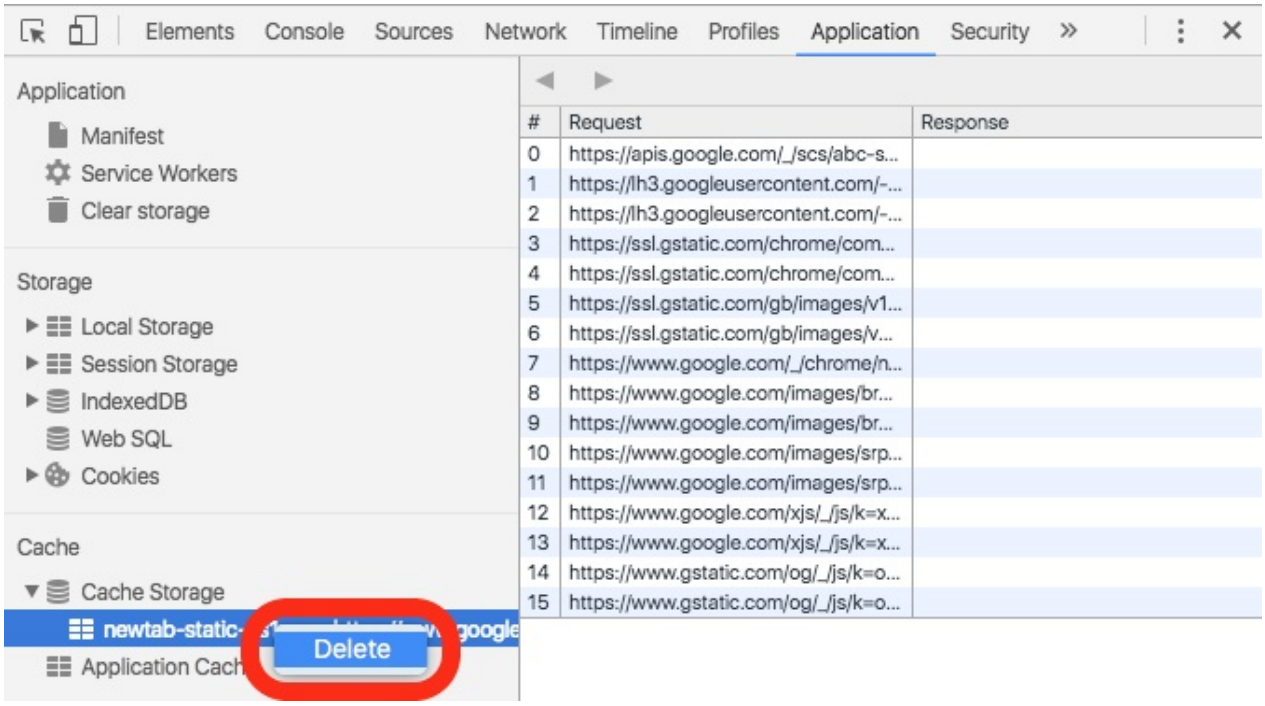


See the MDN article on the Storage Inspector for more information.

Clear the service worker cache

Chrome

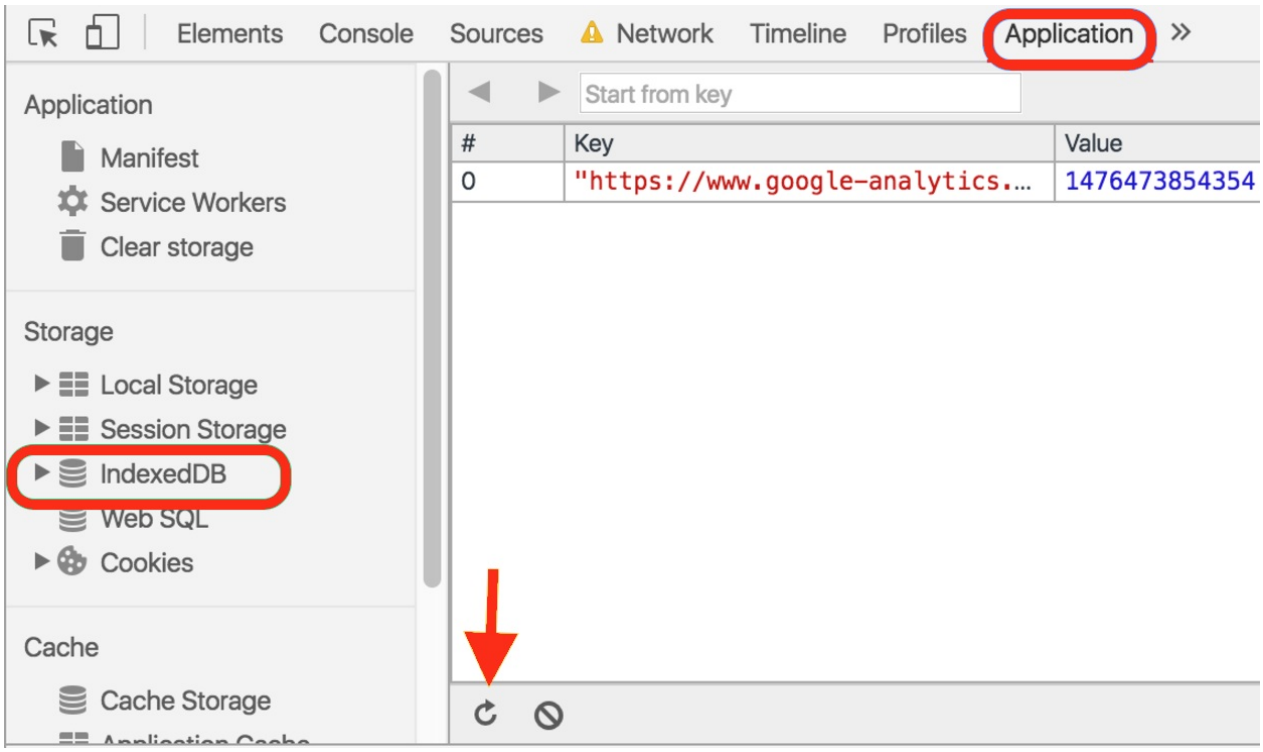
Go to Cache Storage in **DevTools**. In the navigation pane, expand **Cache Storage,**, right-click the cache name and then select **Delete**.



Check IndexedDB

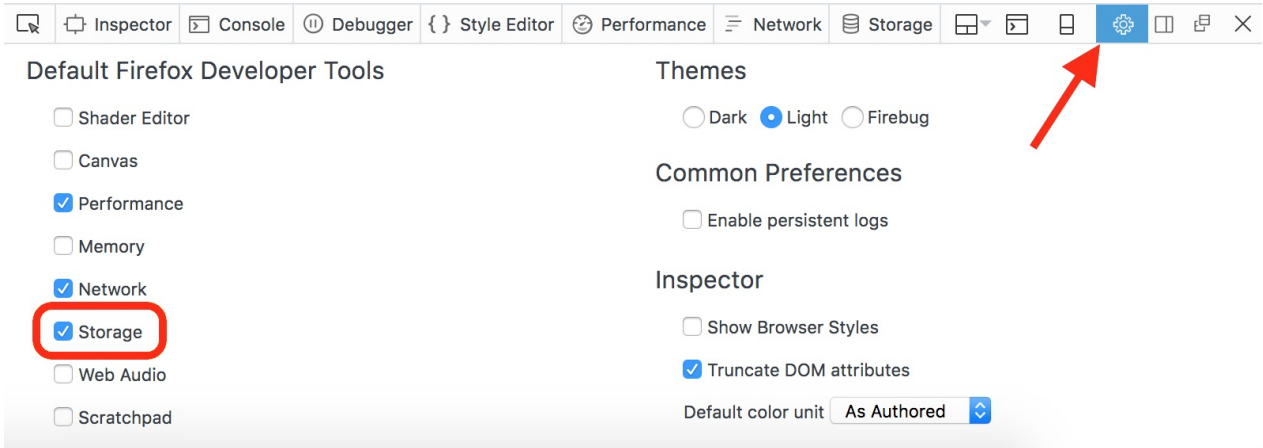
Chrome

In **DevTools**, navigate to the **Application** tab. Select **IndexedDB**. You may need to click Reload to update the contents.

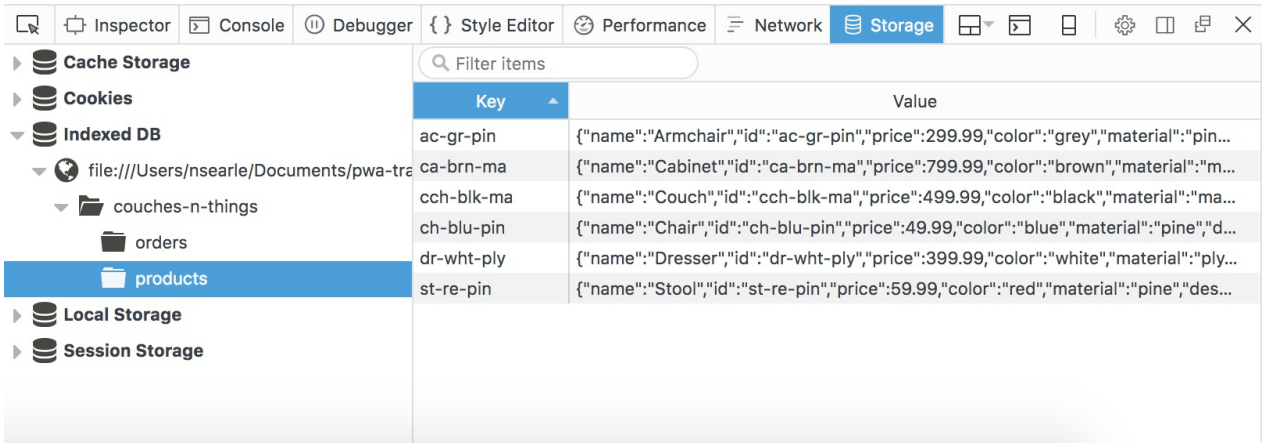


Firefox

Open the Toolbox and click the Settings icon to open **Settings**. Under **Default Firefox Developer Tools,** check **Storage**.



Open the **Storage** panel and expand the **Indexed DB** node. Select a database, object store, or index to see its contents.



Clear IndexedDB

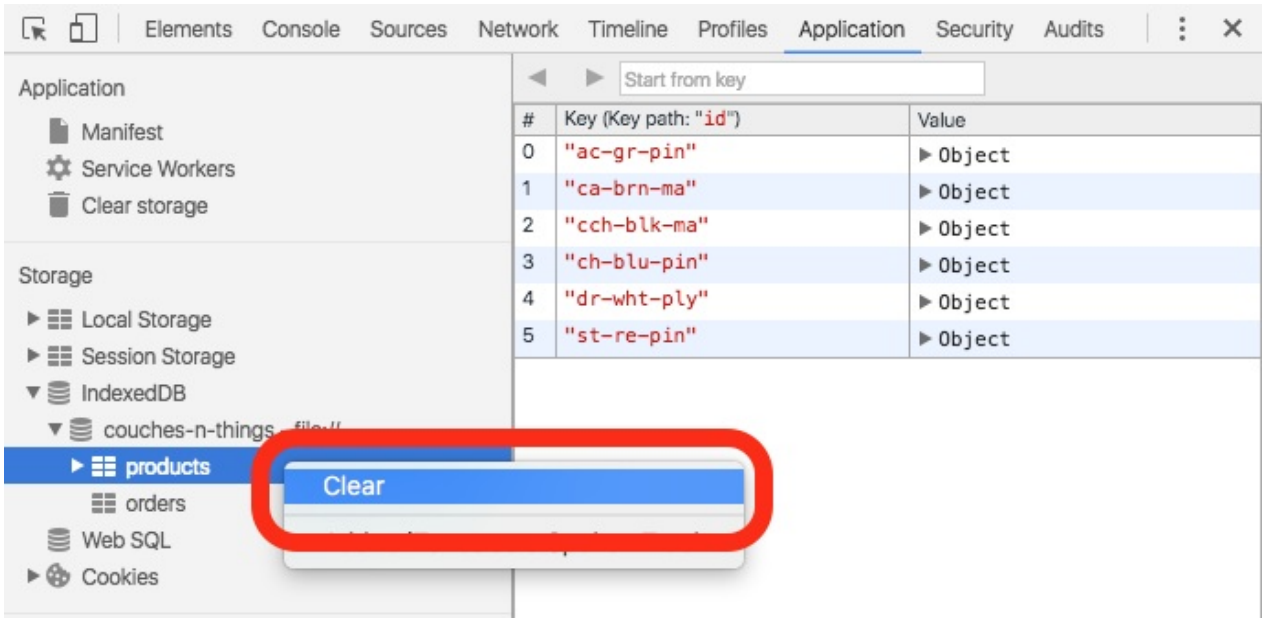
In all browsers that support IndexedDB, you can delete a database by entering the following in the console:

```
indexedDB.deleteDatabase('database_name');
```

Where `database_name` is the name of the database to delete.

Chrome

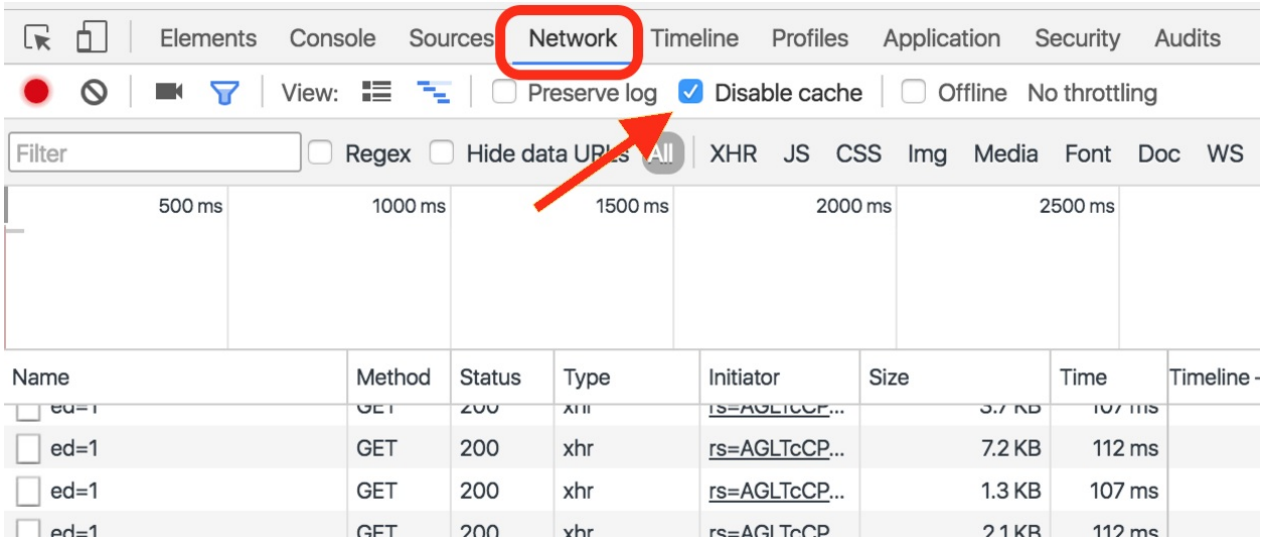
Open IndexedDB in **DevTools**. In the navigation pane, expand **IndexedDB**, right-click the object store to clear, and then click **Clear**.



Disable HTTP Cache

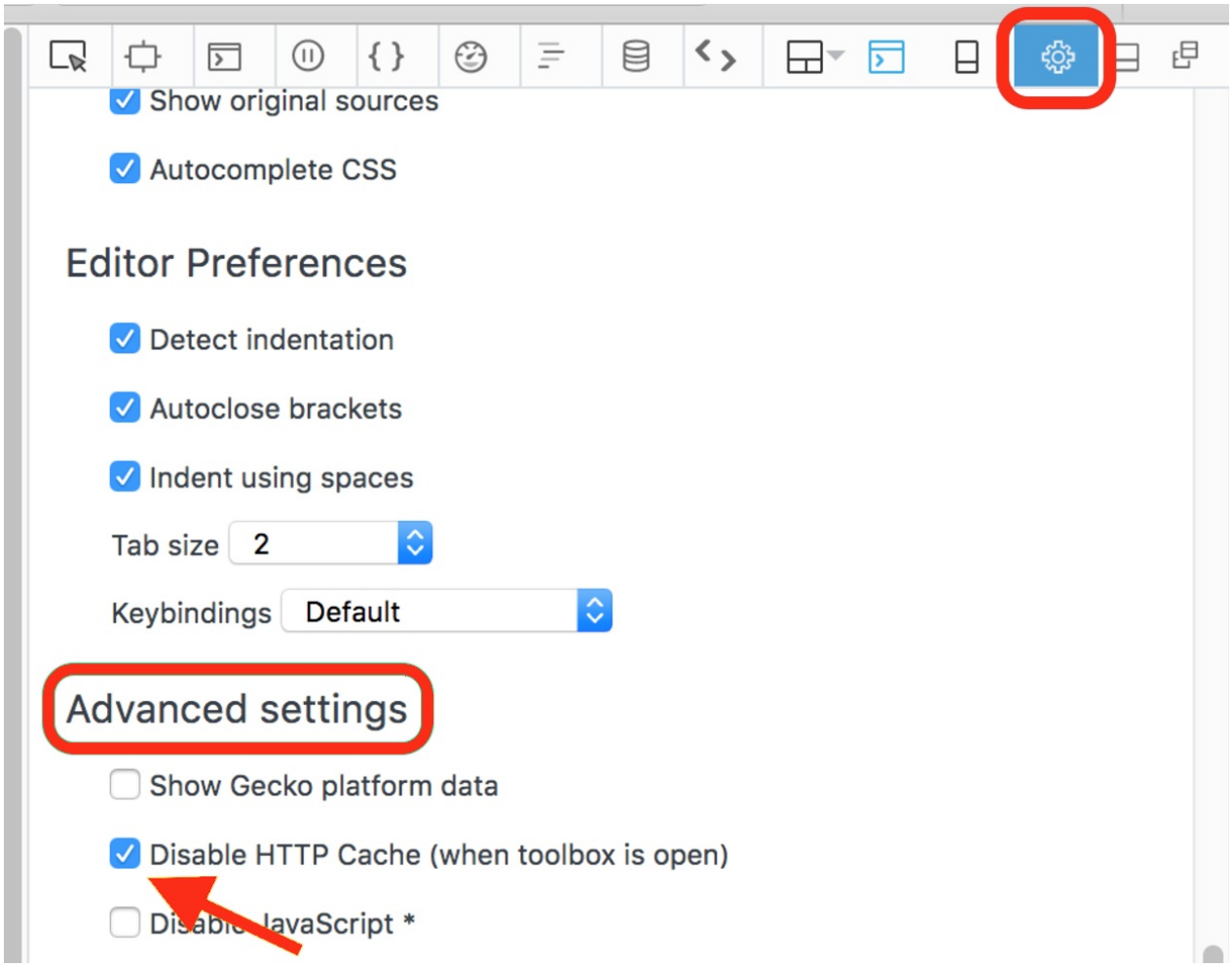
Chrome

Open DevTools and open the **Network** panel. Check the **Disable cache** checkbox.



Firefox

Open the Toolbox and click the Settings icon to open the **Settings**. Under **Advanced settings**, select **Disable HTTP Cache**.



Further reading

Chrome

- Debugging Progressive Web Apps

Safari

- Web Development Tools
- Safari Web Inspector Guide

Firefox

- Opening Settings

Opera

- Opera Dragonfly documentation

Internet Explorer

- Using the Debugger Console
- Debugging Script with the Developer Tools
- Using the Console to view errors and debug

Debugging and FAQ for Progressive Web Apps

This document attempts to answer frequently asked questions about progressive web apps (PWAs) and the PWA training labs:

Debugging Issues

- Issue: changes don't occur when I update my code
- Issue: developer tools doesn't show updates in the cache or IndexedDB
- Issue: Node server won't start
- Issue: srcset is loading an extra image
- Issue: I can't run lighthouse from the command line
- Issue: curl isn't working

FAQ

- What does this console log mean?
- How secure are IndexedDB and the service worker cache?
- How do query parameters work with the Cache API?
- What are the storage limitations of IndexedDB and the service worker cache?
- What's the difference between the browser's original cache and this new service worker cache?

If you have additional questions after reading this FAQ, please let David or Nick know.

Tips

- Use only stable browser versions. Don't use an unstable browser build such as Canary. These can have all types of bugs that are hard to diagnose, particularly with evolving API's like the service worker.
- When developing, disable the HTTP cache. Browsers automatically cache files to save data. This is very helpful for performance, but can lead to issues during development, because the browser might be serving cached files that don't contain your code changes.

Common lab issues

General

Issue: changes don't occur when I update my code

If you are making code changes, but not seeing the intended results, try the following:

- Check that you are in the right directory. For example if you are serving **localhost:8000/lab-foo/app/**, you might be accidentally editing files in **lab-foo/solution/** or **lab-bar/app**.
- Check that the HTTP cache is disabled. Even if you are not caching files with a service worker, the browser might be automatically caching the file on which you are working. If this occurs, the browser runs the cached file's code instead of your updated code.
- Similarly, check if an old service worker is actively serving files from the service worker cache, then unregister it.

Issue: developer tools don't show updates in the cache or IndexedDB

Sometimes when you run some code that caches resources with the Cache API or stores objects in IndexedDB, you won't see any changes in the data displayed in developer tools. This does not necessarily mean that the data was not stored. The developer tools user interface does not automatically update with every change. Depending on your browser, there may be a way to update the developer tools UI (see this example of updating IndexedDB UI in Chrome) to reflect recent changes. If that is not available, you can reload the page.

Setting up

Issue: Node server won't start

There could be more than one cause for Node issues.

First, make sure that Node is actually installed. To confirm this, run the following command:

```
node -v
```

This logs the current version of Node installed. If you don't see a log (indicating that Node is not installed) or the version logged is less than v6, return to the Setting up the labs and follow the instructions for installing Node and the server package (`http-server`).

If Node is installed and you are still unable to start the server, the current port may be blocked. You can solve this by changing ports. The port is specified with the `-p` flag, so the following command starts the server on port 8001:

```
http-server -p 8001 -a localhost -c 0
```

You can generally use any port above 1024 without privileged access.

Note: Don't forget to then use the correct port when opening your app in the browser (e.g., `http://localhost:8001/`).

Responsive Images

Issue: srcset is loading an extra image

You might notice in the Responsive Images lab that the network panel in developer tools shows multiple images loading when `srcset` is used. If a larger version of an image is available in the browser (HTTP) cache, some browsers might load that image even if it is not the one specified by `srcset` . Because the browser already has a higher resolution image stored locally, it does not cost any data to use the better quality image.

You can confirm that this is the case in some browsers' developer tools by inspecting the network requests. If a file is coming from the browser cache, it is usually indicated. For example:

- In Chrome, the **Size** property of the file says "from memory cache".
- In Firefox, the **Size** is "0 B" (for zero bytes transferred).
- In Safari, there are explicit **Cache** and **Transferred** properties, specifying if the file was cached and how much data was transferred over the network, respectively.

For simplicity in the lab, make sure the HTTP cache is disabled in developer tools.

Lighthouse

Issue: I can't run lighthouse from the command line

Lighthouse requires Node v6 or greater. You can check your Node version with the following command:

```
node -v
```

If you are using less that Node v6, update to the current Long Term Support (LTS) version of Node, or install a tool like Node Version Manager. See Setting up the labs for instructions.

Note: It may be possible to use the `--harmony flag with Node v4 or v5.`

Integrating Web Push

Issue: curl is not working

There could be more than one reason curl isn't working for you.

- **Windows users:** Curl is not available by default on Windows, so if you're using Windows you can skip that step (curl is just for testing push messages in development without having to write a node script). The lab walks you through sending a push message using node in the next step. If you're set on using curl, you can download it, unzip the package, and use it from the command line (no install necessary)
- Spacing and/or quotations are incorrect. Curl is very picky about syntax. Make sure the spacing and quotations are exactly the same as in the provided example.
- The Endpoint URL and/or server key are incorrect. Make sure you have correctly copied the full Endpoint URL from the page after enabling push messaging. The server key must be copied from your project on Firebase

Frequently Asked Questions

What does this console log mean?

When working with the service worker, you may see a console log similar to the following:

```
Service Worker termination by a timeout timer was canceled because DevTools is attache
d.
```

This is not an error. For the purposes of our labs, you can ignore this log.

Under normal conditions, the browser terminates service workers when they are not in use in order to save resources. This does not delete or uninstall a service worker, but simply deactivates it until it becomes needed again. If developer tools are open, a browser might not terminate the service worker as it would normally. This log lets the developer know that the service worker termination was cancelled.

How secure are IndexedDB and the service worker cache?

IndexedDB and the service worker caches are stored unencrypted, but access is restricted by origin (similar to cookies and other browser storage mechanisms). For example, **foo.com** should not be able to access IndexedDB or caches from **bar.com**. However, if an attacker successfully performs a cross-site scripting (XSS) attack, then they gain access to all origin storage, including IndexedDB and the service worker cache, as well as cookies. You should never store user passwords locally (or even server-side), but you could store session tokens in IndexedDB with similar security to cookies (although cookies can be set to HTTP-only).

How do query parameters work with the Cache API?

The Cache API lets you store request/response pairs in the browser, where the request is the key and the response is the value. Responses are retrieved from the cache by searching for the corresponding request. Let's look at an example.

The request is a path or URL, which can be arbitrarily set by the developer using `caches.put`.

```
fetch('./example/resource.json').then(function(response) {
  caches.open('exampleCache').then(function(cache) {
    cache.put('http://example.com/resource.json', response);
})
```

Here we fetch an example JSON file and store it in `exampleCache`. We have set the key for that file to `http://example.com/resource.json`.

To fetch the JSON file, we would pass that key into `caches.match`. This method takes a request as the first argument, and returns the first matching response in the cache. For example, to retrieve the response we stored previously:

```
caches.match('http://example.com/resource.json')
.then(function(response) {
  return response;
})
```

What are the storage limitations of IndexedDB and the service worker cache?

Storage limitations for origin storage mechanisms (like IndexedDB, service worker caches, cookies, etc.) are browser dependent. From Offline Storage for Progressive Web Apps:

- In Chrome and Opera, storage is per origin (rather than per API). IndexedDB and the Cache API store data until the browser quota is reached. Apps can check how much quota they're using with the Quota Management API.

- Firefox has no limits, but will prompt the user after 50MB of data is stored.
- Mobile Safari has a 50MB max.
- Desktop Safari has unlimited storage,but prompts the user after 5MB.
- IE10+ has 250MB but prompts the user at 10MB.

Where *per origin* means that API's like `localStorage` would be sharing storage space with API's like IndexedDB. There is also a Persistent Storage API, that allows storage to become permanent.

What's the difference between the browser's original cache and this new service worker cache?

There are two types of cache in the browser: browser-managed cache and application-managed cache.

- Browser-managed caches (often called "HTTP cache" or "browser cache") are a temporary storage location on your computer for files downloaded by your browser to display websites. Files that are cached locally include any documents that make up a website, such as HTML files, CSS style sheets, JavaScript scripts, as well as graphic images and other multimedia content. This cache is managed automatically by the browser and is not available offline.
- Application-managed caches are also a temporary storage location on your computer for files. However, these caches are created using the Cache API independent of the browser-managed caches. This API is available to applications via `window.caches` and via the service worker. Application-managed caches hold the same kinds of assets as a browser cache but are accessible offline (e.g., by the service worker to enable offline support). This cache is managed by developers who implement scripts that use the Cache API to explicitly update items in named cache objects.

Caching is a good technique that you can use when building your app, as long as the cache you use is appropriate for each resource. Both cache implementations have similar performance.