INSTALLING PWAS

& Technology Overview

To qualify as a Progressive Web App you must have the following.

We have seen the first 3:

- A HTML page
- A Service Worker
- To be served securely over HTTPS

We also will need

- A Manifest
- An Icon (required by some browsers)

Apart from the minimum requirements we will also briefly look at other features that can be an important part of PWAs

Notifications

Push Messages

Install Banners

Background Sync

IndexDB

Manifest

On requirement for your site to qualify as a Progressive Web App is to have a manifest file. This is a JSON file that contains the meta data required for your site to act as an application, and **to be installable**.

I.e. The ability to **A**dd to (**2**) **H**ome **S**creen is often considered an important feature of a PWA.

This is sometimes known as:

A2HS

This file is frequently called **manifest.json**

The standard suggests a file extension of .**webmanifest**

E.g. **manifest.webmanifest**

You then link to the manifest in your HTML

```
<link rel="manifest" href="manifest.webmanifest">
```

At a minimum, the JSON object must contain the following properties:

background_color

Mainly this is used for the background of the splash screen when loading the app.

Although it can be used in other context too.

display

When running as an app (i.e. not directly in the browser) you can specify how to show the web pages on the device.

Primarily you decide whether the device UI (such as the status bar) should be visible.

For apps can set it to **fullscreen** (no UI) or **standalone** (may show some UI like the status bar)

icons

Here you specify the images to use as icons in various contexts (e.g. the icon that appears on the homescreen). This will be an array of objects describing the icon at different sizes (for responsive sites).

```
"icons": [{
      "src": "icon/icon-sml.png",
      "sizes": "64x64",
      "type": "image/png"
    }
,
    {
      "src": "icon/icon.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ]
```

NOTE: To have your app installable on a device you may* need to provide an icon graphic.

*Browsers like Chrome require it.

name
short_name

Versions of your apps name that can be used in various contexts (e.g. the **short_name** can be used with the icon on the home screen if there isn't enough room for the full version, i.e. **name**).

start_url

This is the relative path of the page you want to use when you launch your site.

I.e. it is relative to the manifest file.

This will often be your main index.html page. But in some cases you may want a different main page for the installed PWA version of your site.

```
{
  "background_color": "purple",
  "description": "Shows random fox pictures. ",
  "display": "fullscreen",
  "icons": [
    {
      "src": "icon/fox-icon.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ],
  "name": "Awesome fox pictures",
  "short_name": "Foxes",
  "start_url": "/pwa-examples/a2hs/index.html"
}
```

https://developer.mozilla.org/en-US/Apps/Progressive/Add_to_home_screen

The assets needed by your app will have to be downloaded by a service worker.

Other Features

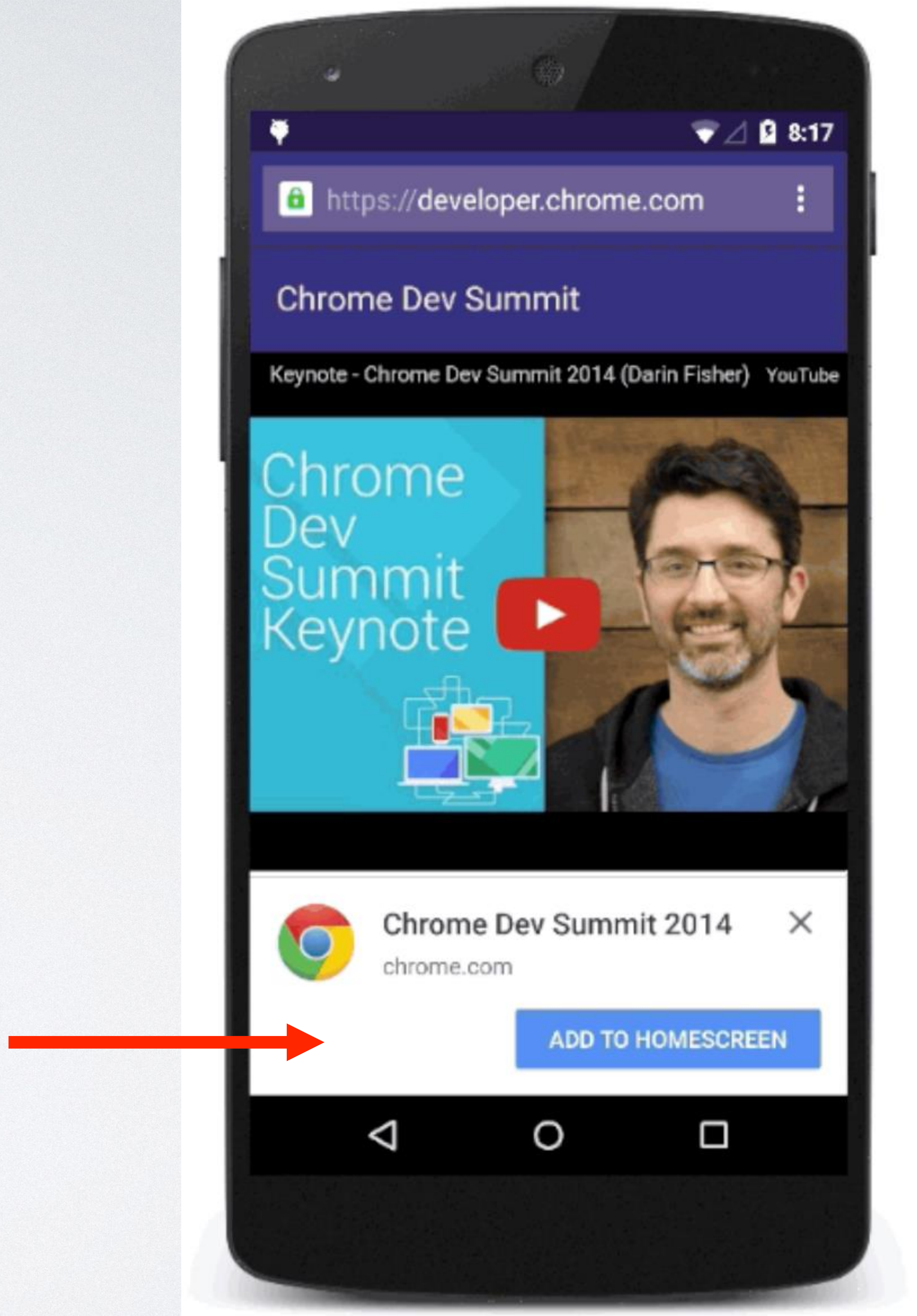
While not required, there are man other features that make up a PWA experience.

Install Banners

Many devices will detect your web page is an app and display a banner asking if you want to save to the home screen.

This is done automatically for you.

Your JavaScript can detect what they chose to do with an event handler.


8:17
https://developer.chrome.com
Chrome Dev Summit
Keynote - Chrome Dev Summit 2014 (Darin Fisher)
YouTube
Chrome Dev Summit Keynote
Chrome Dev Summit 2014
chrome.com
ADD TO HOMESCREEN

The following code just displays messages on the console depending on whether you wanted to install the app or not.

There is a **userChoice** property of the event object that is passed to the **beforeinstallprompt** event handler that we can attach to the **window**.

This event handler is fired ***before the banner is shown***.

userChoice contains a Promise that resolves (when the user eventually interacts with the banner) to an object containing an **outcome** property.

This property contains a string describing the user's choice. E.g. if the user just closed the *Install Banner* then **outcome** contains the string 'dismissed'.

```
window.addEventListener('beforeinstallprompt', function(e) {

  e.userChoice.then(function(choiceResult) {

    console.log(choiceResult.outcome);

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

https://developers.google.com/web/fundamentals/app-install-banners/

The reason the event handler is fired ***before*** the prompt is shown (rather than after it appears, or whenever the user responds ot the banner) is because this gives us an opportunity to stop the banner being shown in the first place. (e.g. if the user dismisses the banner its a good idea not to keep showing the banner every time the user visits subsequently).

E.g. you can stop the prompt from appearing.

```
window.addEventListener('beforeinstallprompt', function(e) {
  console.log('beforeinstallprompt Event fired');
  e.preventDefault();
  return false;
});
```

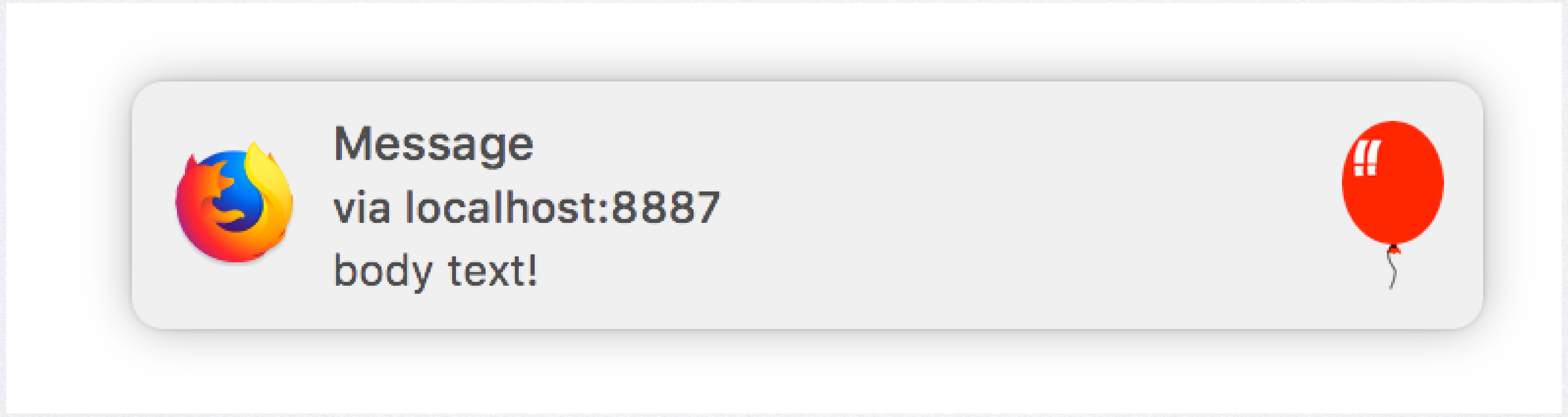
https://developers.google.com/web/fundamentals/app-install-banners/

In your manifest file you can even suggest alternative apps in the app store to appear in the banner.

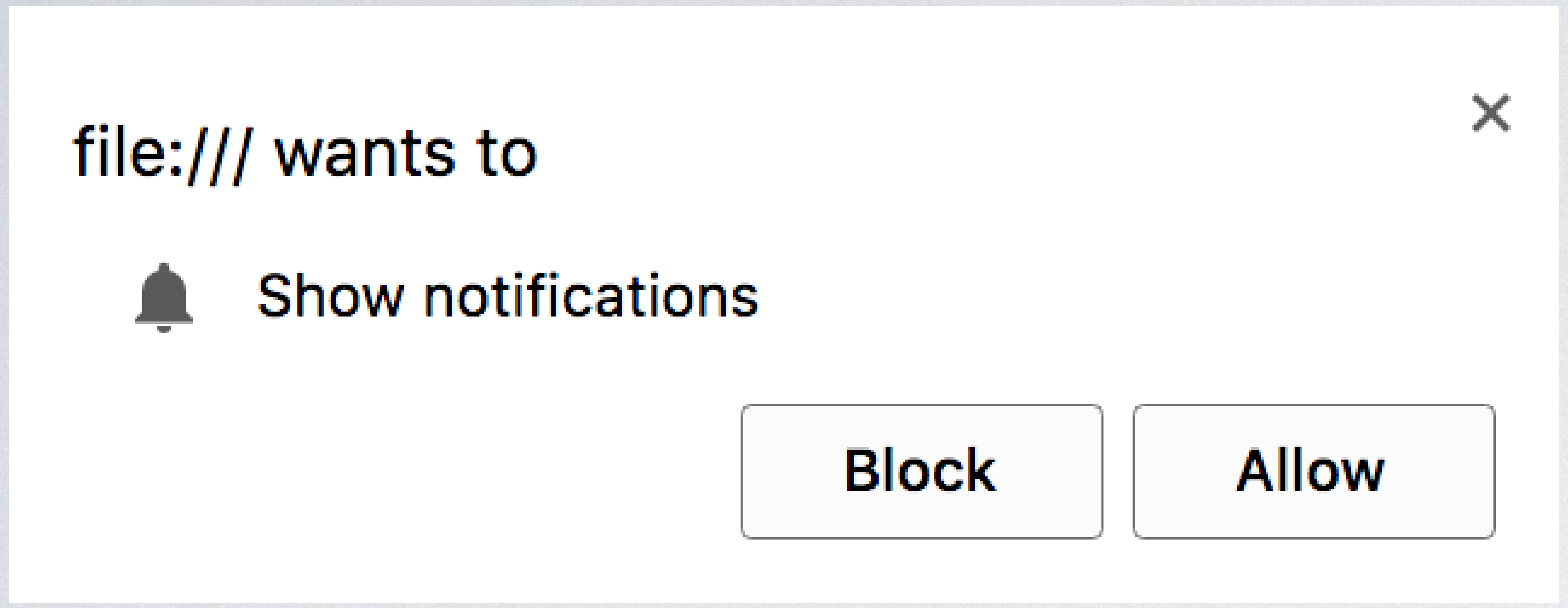
```
"prefer_related_applications": true,
"related_applications": [
  {
  "platform": "play",
  "id": "com.google.samples.apps.iosched"
  }
]
```

Notifications

JavaScript allows you to access the **notification** functionality of the device your web page is running on (assuming the device supports the API, and the user has granted your site permission to send notifications).



Message
via localhost:8887
body text!

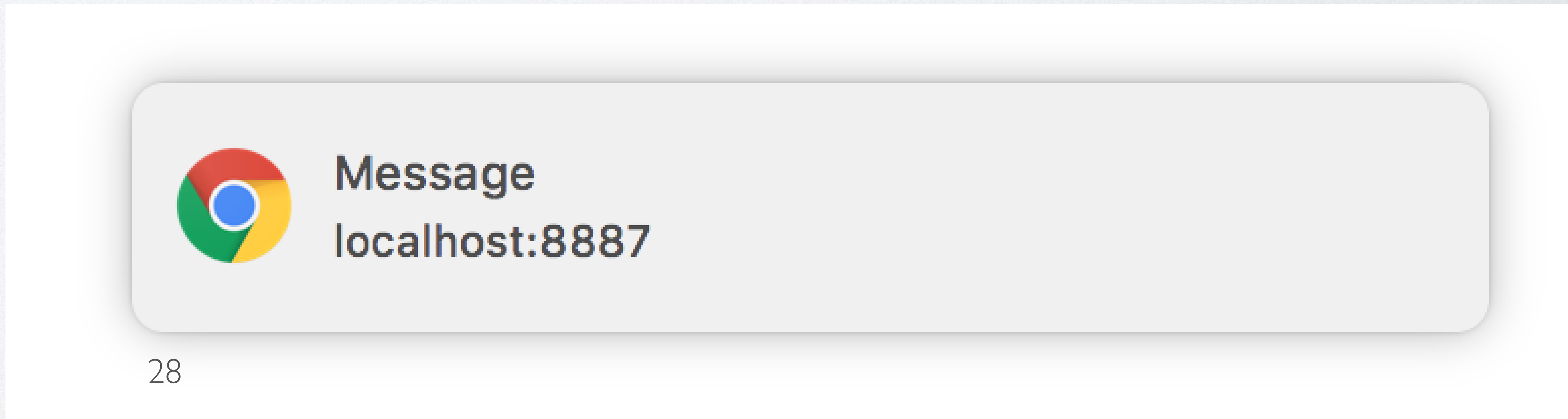

file:/// wants to

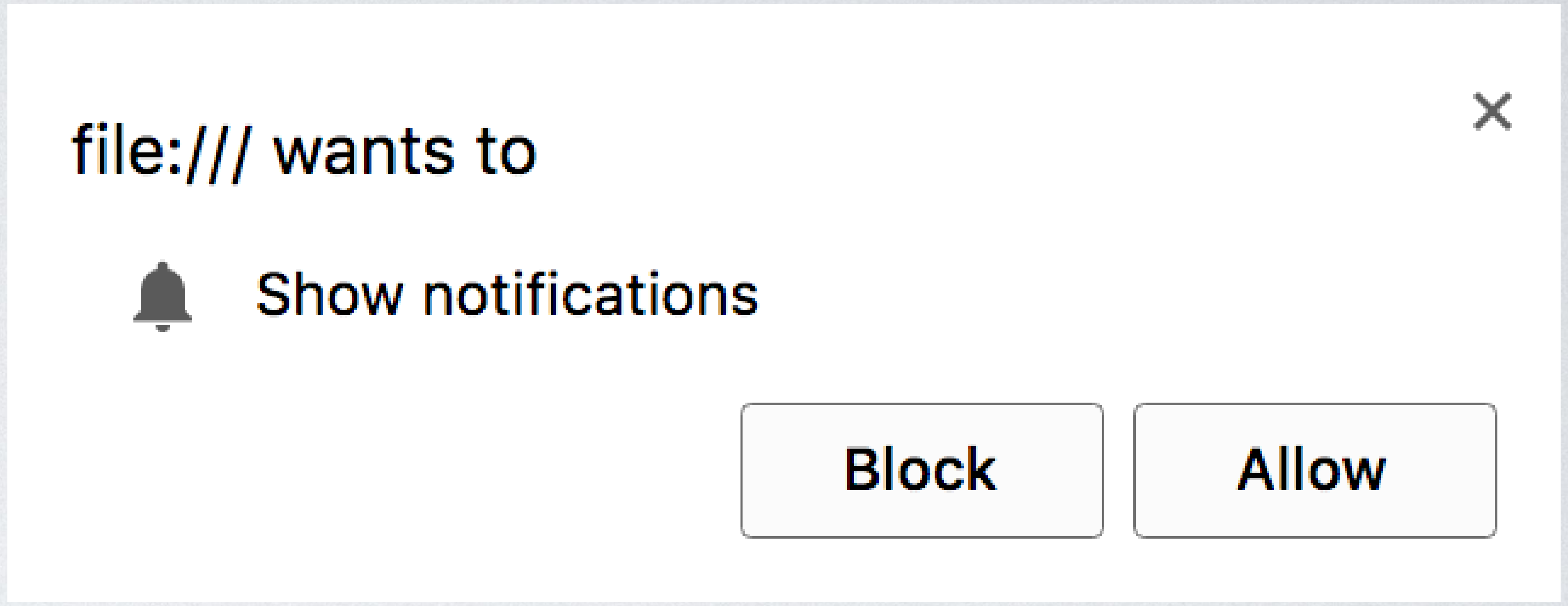
Show notifications

Block
Allow

status stores whether the user has allowed the use of notifications.

```
Notification.requestPermission().then(function(status) {

    if (status === "granted")
    {
    var n = new Notification('Message');

    }
});
```


Message
localhost:8887


file:/// wants to

Show notifications

The **status** can be:

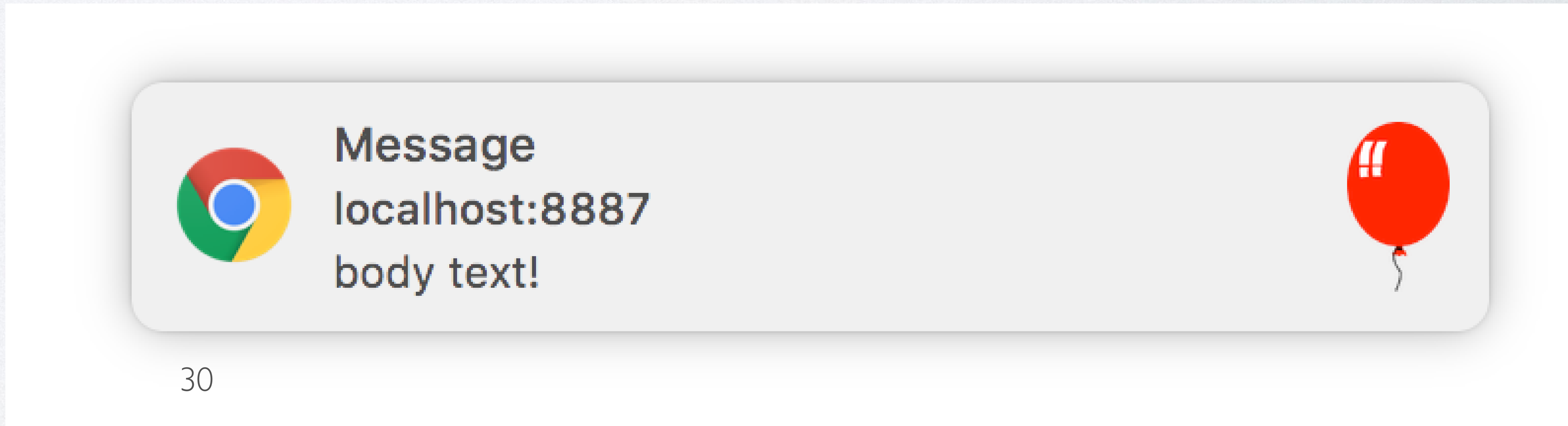
default
User hasn't explicitly given permission for you to send notifications. Notifications can't be sent.

granted
User has been asked and has given permission for you to send notifications.

denied
User has been asked and has not given permission for you to send notifications.

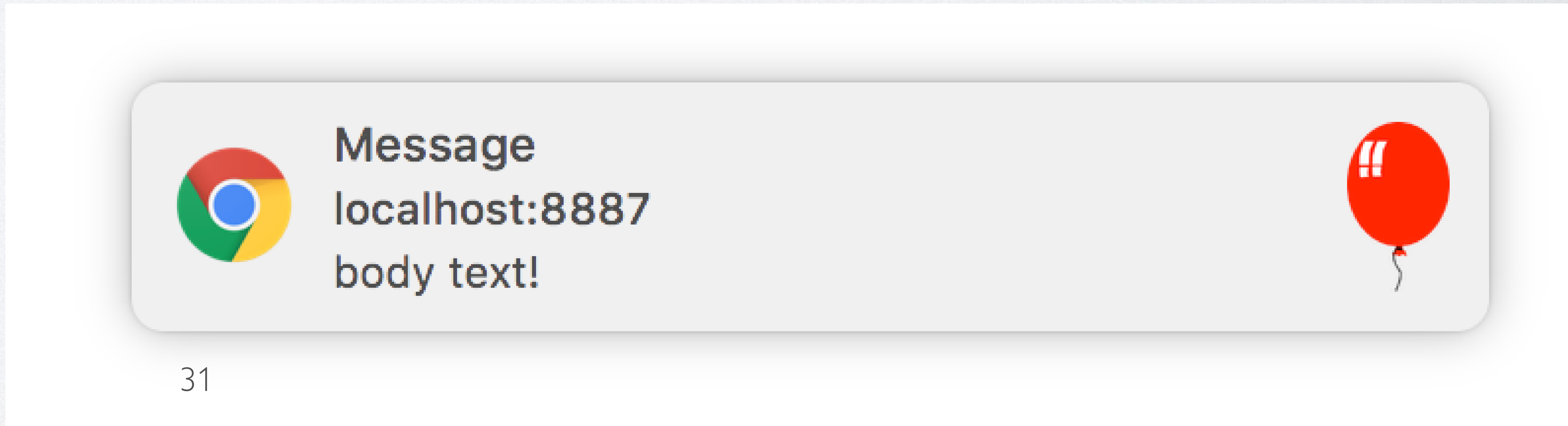
You can supply further options to the **Notification** constructor in an options object.

```
Notification.requestPermission().then(function(status) {

   if (status === "granted")
   {
   var n = new Notification('Message', {
      body: 'body text!',
      icon: 'red-balloon.png'
   });

   }
});
```


Message
localhost:8887
body text!

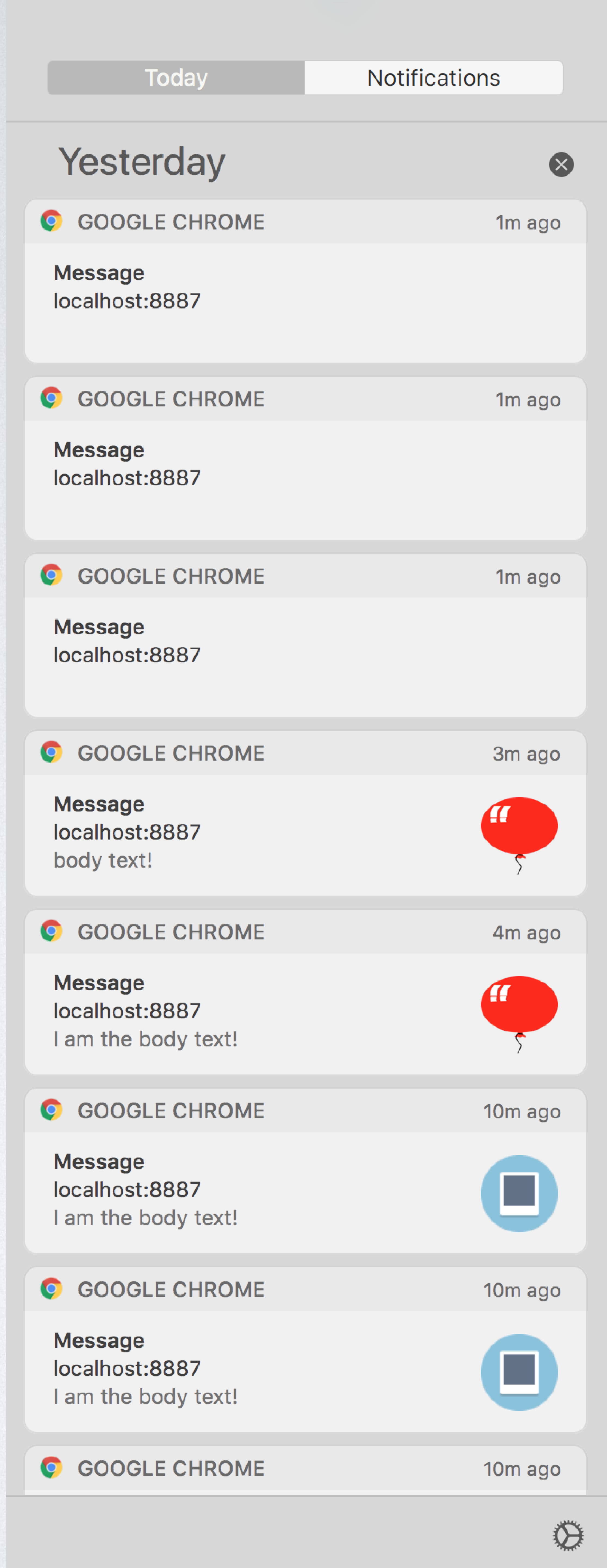
Here we provide the text of the message and the url of an image we want to appear.

```
Notification.requestPermission().then(function(status) {

    if (status === "granted")
    {
    var n = new Notification('Message', {
        body: 'body text!',
        icon: 'red-balloon.png'
    });

    }
});
```



Message
localhost:8887
body text!

These notifications are supported by your OS/Device and not just your browser.



Yesterday

GOOGLE CHROME 1m ago
Message
localhost:8887

GOOGLE CHROME 1m ago
Message
localhost:8887

GOOGLE CHROME 1m ago
Message
localhost:8887

GOOGLE CHROME 3m ago
Message
localhost:8887
body text!

GOOGLE CHROME 4m ago
Message
localhost:8887
I am the body text!

GOOGLE CHROME 10m ago
Message
localhost:8887
I am the body text!

GOOGLE CHROME 10m ago
Message
localhost:8887
I am the body text!

GOOGLE CHROME 10m ago

There are many other options available in the Notification API. Many are device (and permissions) dependant. E.g. You can control how the device vibrates when it displays the message.

```
function displayNotification() {
  if (Notification.permission == 'granted') {
    navigator.serviceWorker.getRegistration().then(function(reg) {
      var options = {
        body: 'Here is a notification body!',
        icon: 'images/example.png',
        vibrate: [100, 50, 100],
        data: {
          dateOfArrival: Date.now(),
          primaryKey: 1
        }
      };
      reg.showNotification('Hello world!', options);
    });
  }
}
```

Can set a vibrate pattern for when the notification appears.

Can be used to identify the notification when you click on it.

https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications

In a service worker you can access the Notification API through the **self.registration** object.

```
self.registration.showNotification('Hello world!')
```

Push API

The push API allows a server to interact with a user outside of the web site/web application itself.

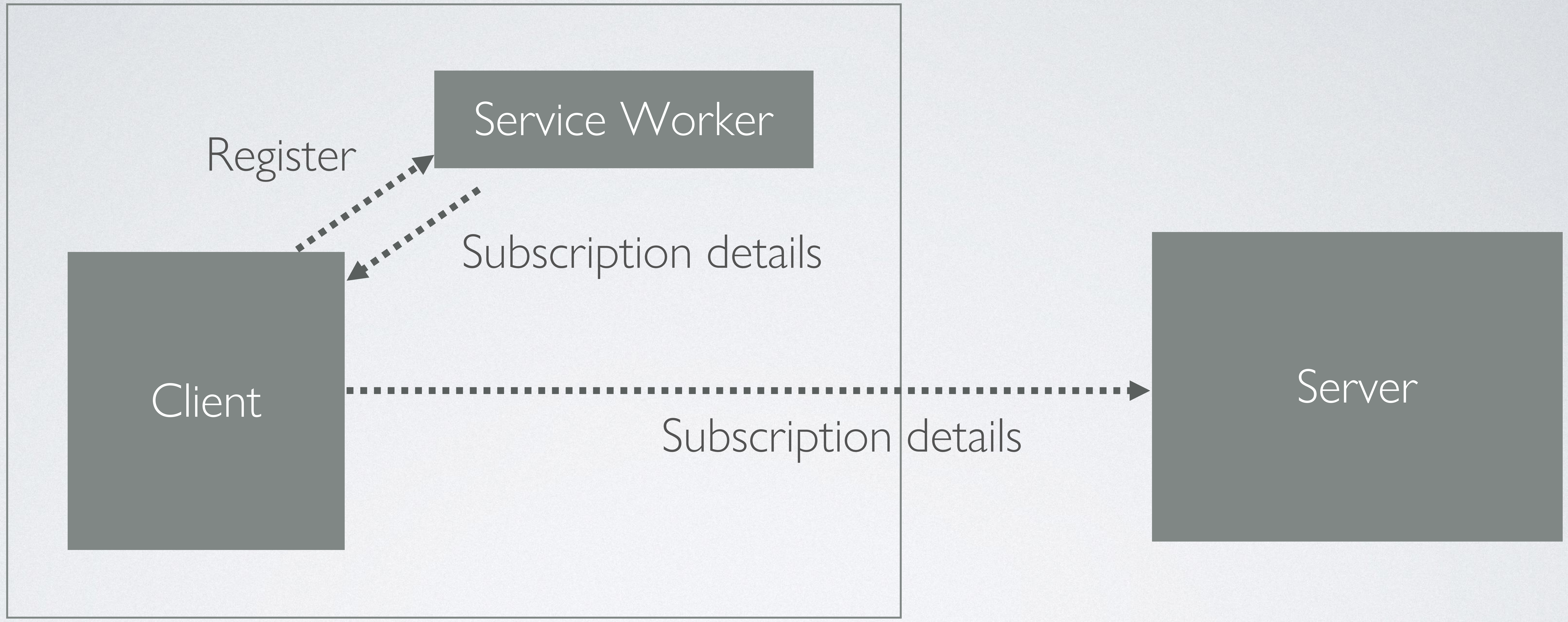
E.g. the server can interact with your device's notification system.

(**Note:** the user will be automatically asked if they want to enable this feature.)

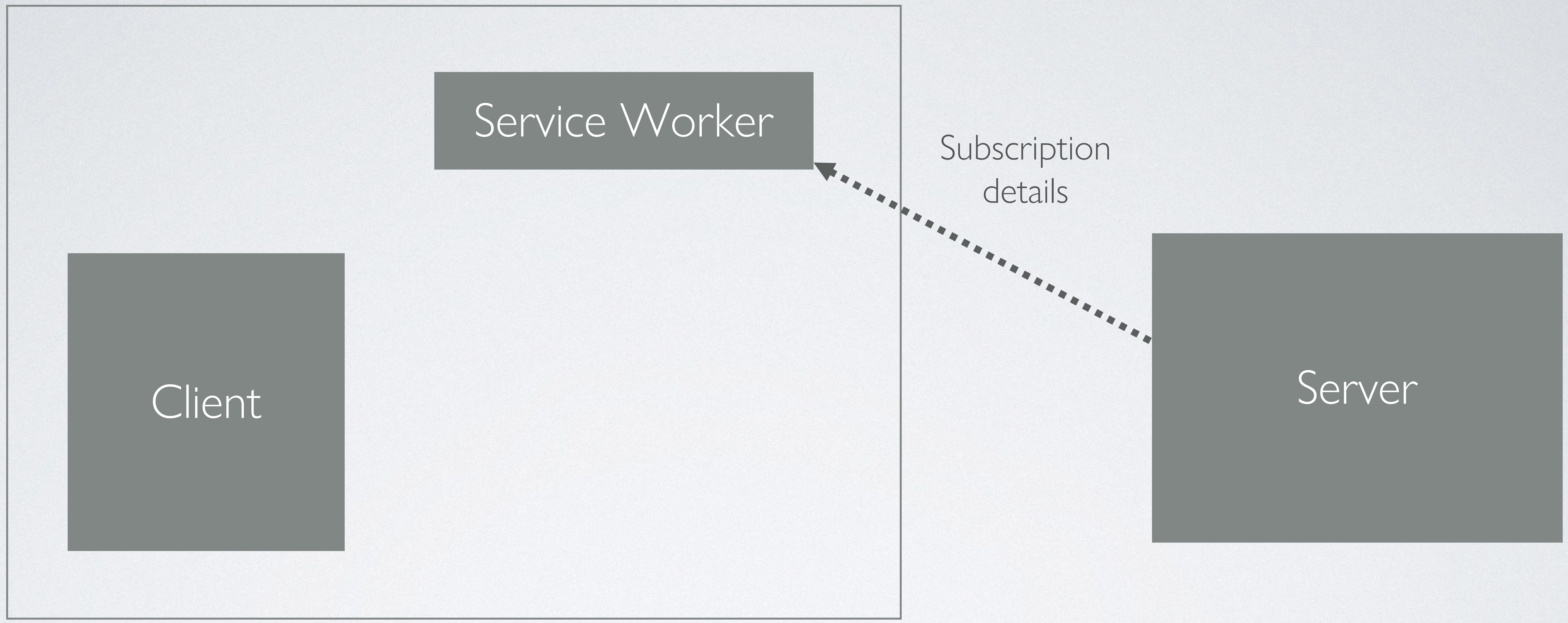
This is implemented by your service worker. The service worker can lie dormant after you have finished with the web page that registered it. It can then reactivate in order to respond to the server **push**ing messages to it.

(Since you may no longer be using the web page the service worker can use the notification API to get your attention.)

Your application must **subscribe** to the server in order to allow it access to your service worker.



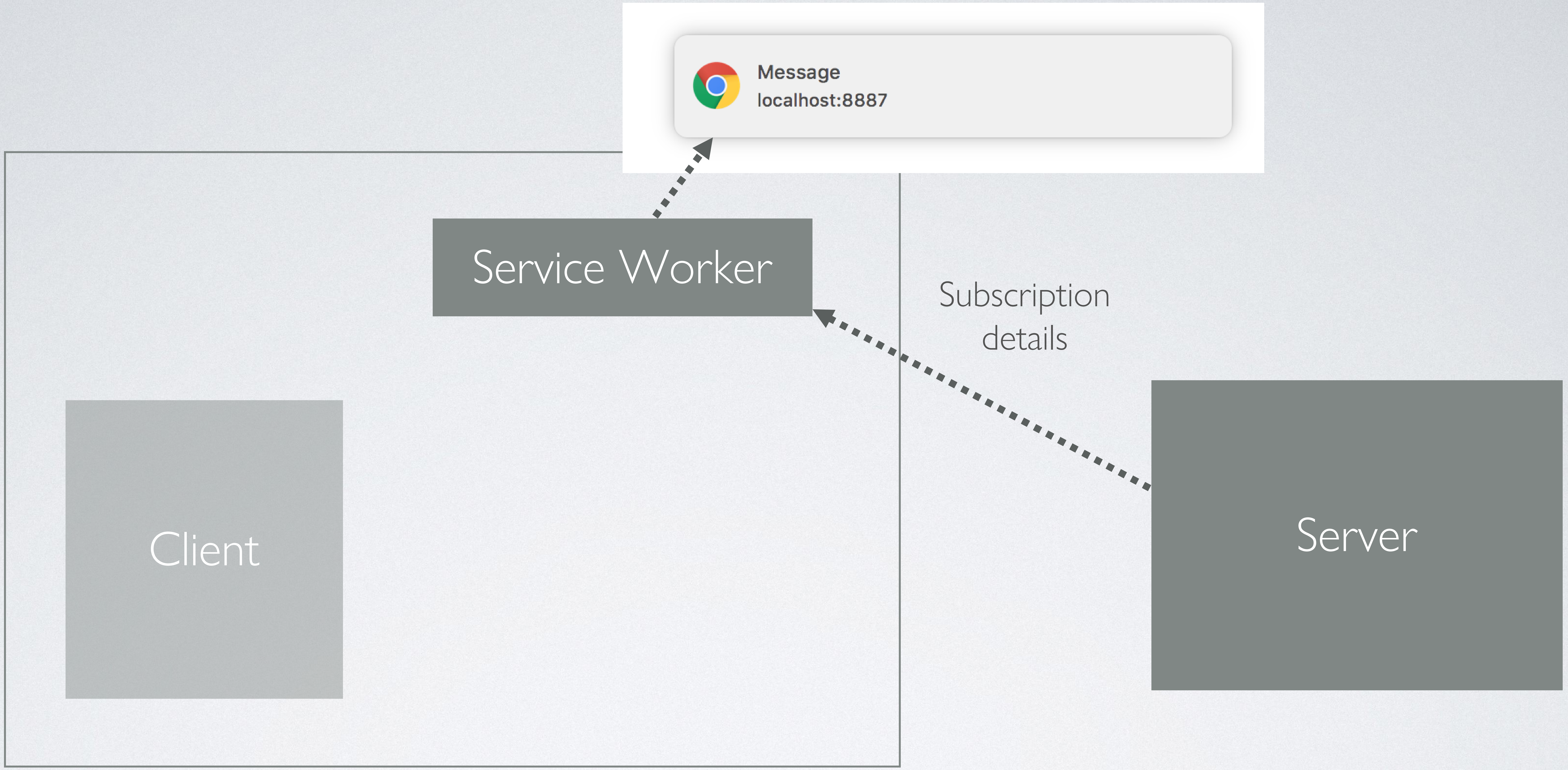
Your web pages can get subscription details from their service workers which they can then send to the server. These unique details are used to ensure the **push** requests are legitimate.



The server can send data to your service worker at any time using the subscription details you sent it in order to prove its identity.

Your server is likely to push messages to all the clients that subscribe to it.

E.g. a news app will send information about breaking news.



The Push API can be used in conjunction with Notification API. The service worker can reactivate regardless of whether the page using it is open.

For this to work the server must be able to contact your device.

If you register with your service worker the service worker can send back subscription details including a unique endpoint (URL) the server can use.

The details look something like the following:

```
{"endpoint":"https://fcm.googleapis.com/fcm/send/dpH5lCsTSSM:APA91bHqjZxM0VImWWqDRN7U0a3AycjUf4O-byuxb_wJsKRaKvV
_iKw56s16ekq6FUqoCF7k2nICUpd8fHPxVTgqLunFeVeB9lLCQZyohyAztTH8ZQL9WCxKpA6dvTG_TUIhQUFq_n",
"keys": {
    "p256dh":"BLQELIDm-6b9Bl07YrEuXJ4BL_YBVQ0dvt9NQGGJxIQidJWHPNa9YrouvcQ9d7_MqzvGS9Alz60SZNCG3qfpk=",
    "auth":"4vQK-SvRAN5eo-8ASlrwA=="
    }
}
```

https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications

The following code shows how you can register for a service worker and have it return a Promise. The promise is resolved with a **ServiceWorkerRegistration** object that also contains the subscription details.

I.e. if **reg** is the **ServiceWorkerRegistration** object then the subscription details are in:

```
reg.pushManager.getSubscription()
```

Note that this function returns a Promise.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('sw.js').then(function(reg) {
    console.log('Service Worker Registered!', reg);

    reg.pushManager.getSubscription().then(function(sub) {
      if (sub === null) {
        // Update UI to ask user to register for Push
        console.log('Not subscribed to push service!');
      } else {
        // We have a subscription, update the database
        console.log('Subscription object: ', sub);
      }
    });
  })
   .catch(function(err) {
    console.log('Service Worker registration failed: ', err);
  });
}
```

Getting subscription details for the push service in your web page.

https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications

The following example is of how a ServiceWorker responds to a push message from a server. In this case it just notifies the user using the OS Notification API.

E.g. in this code sample we add an event listener that checks for incoming push messages. If we receive one it just passes a "Hello World!" message to the user via the regular Notification API.

```
self.addEventListener('push', function(e) {
  var options = {
    body: 'This notification was generated from a push!',
    icon: 'images/example.png',
    vibrate: [100, 50, 100],
    data: {
      dateOfArrival: Date.now(),
      primaryKey: '2'
    },
    actions: [
      {action: 'explore', title: 'Explore this new world',
        icon: 'images/checkmark.png'},
      {action: 'close', title: 'Close',
        icon: 'images/xmark.png'},
    ]
  };
  e.waitUntil(
    self.registration.showNotification('Hello world!', options)
  );
});
```

Responding to a push event from the server (in the Service Worker)

https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications

Background Sync API

Background Sync allows you delay the performance of some action until the device is online (or some other impediment is resolved).

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

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

Client

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

Service Worker

https://developers.google.com/web/updates/2015/12/background-sync

The **doSomeStuff()** function should try an perform some action. It should return a Promise that resolves or rejects based on whether its action is completed or not (e.g. if the device not online it may not be able to complete). If the Promise is rejected then the function call is rescheduled for another time.

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

Service Worker

https://developers.google.com/web/updates/2015/12/background-sync

IndexedDB API

PWAs also have the ability to store persistent data on the client that will be available the next time you launch your app (regardless of whether you are online or not).

We have seen **localStorage** as a way of storing small amounts of data. However, saving and retrieving the storage is synchronous which can slow down your application. You are also restricted in terms of the type (strings) and amount of data you can store.

indexDB gives you the advantage of a database local to your browser/application environment which is asynchronous, allows large amounts of storage, and provides rich query abilities.

We won't be covering it this year but there is more information online.

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

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