Don't miss out on the action at this year's Chrome Dev Summit, happening on Oct 23rd and 24th. Learn more (https://developer.chrome.com/devsummit/).

ウェブアプリへのプッシュ通知の追加

By Matt Gaunt
(https://developers.google.com/web/resources/contributors#mattgaunt)
Matt is a contributor to Web**Fundamentals**

概要

プッシュ メッセージは、ユーザーに改めて訴えかける簡単で効果的な方法です。このコードラボでは、プッシュ通知をウェブアプリに追加する方法を学びます。

学習内容

- プッシュ メッセージ用にユーザーの登録や登録解除を行う方法
- 受信プッシュ メッセージを処理する方法
- 通知を表示する方法
- 通知のクリックに応答する方法

必要なもの

- Chrome 52 以上
- Web Server for Chrome
  (https://chrome.google.com/webstore/detail/web-server-for-chrome/ofhbbkphhbklhfoeikjpcbhemlocgigb)
  またはご利用のウェブサーバー
- テキスト エディタ
- HTML、CSS、JavaScript、Chrome DevTools の基本知識

- サンプルコード（「準備」をご覧ください）

準備

サンプルコードのダウンロード

このコードのサンプルコードを入手するには、次のリンクで zip をダウンロードします。

リンク (https://github.com/googlechrome/push-notifications/archive/master.zip)

または、次の git レポジトリのクローンを作成します。

```
git clone https://github.com/GoogleChrome/push-notifications.git
```

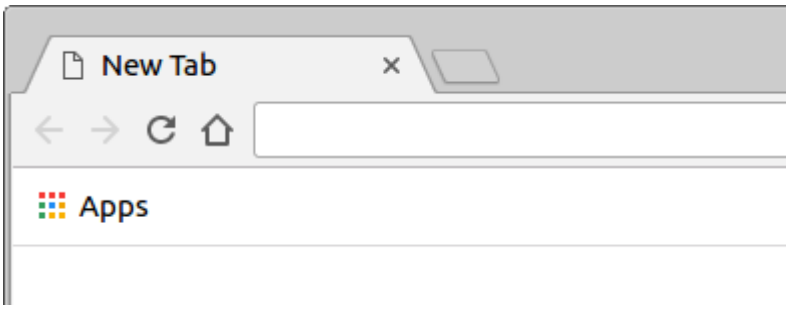
ソースを zip としてダウンロードし、これを解凍すると、ルートフォルダ **push-notifications-master** が作成されます。

ウェブサーバーのインストールと確認

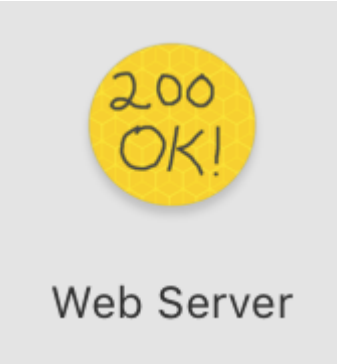
自分のウェブサーバーを使用することができますが、このコードラボは、Chrome Web Server で問題なく動作するように設計されています。このアプリをまだインストールしていない場合は、Chrome ウェブストアからインストールできます。

リンク
(https://chrome.google.com/webstore/detail/web-server-for-chrome/ofhbbkphhbklhfoeikjpcbhemlocgigb)

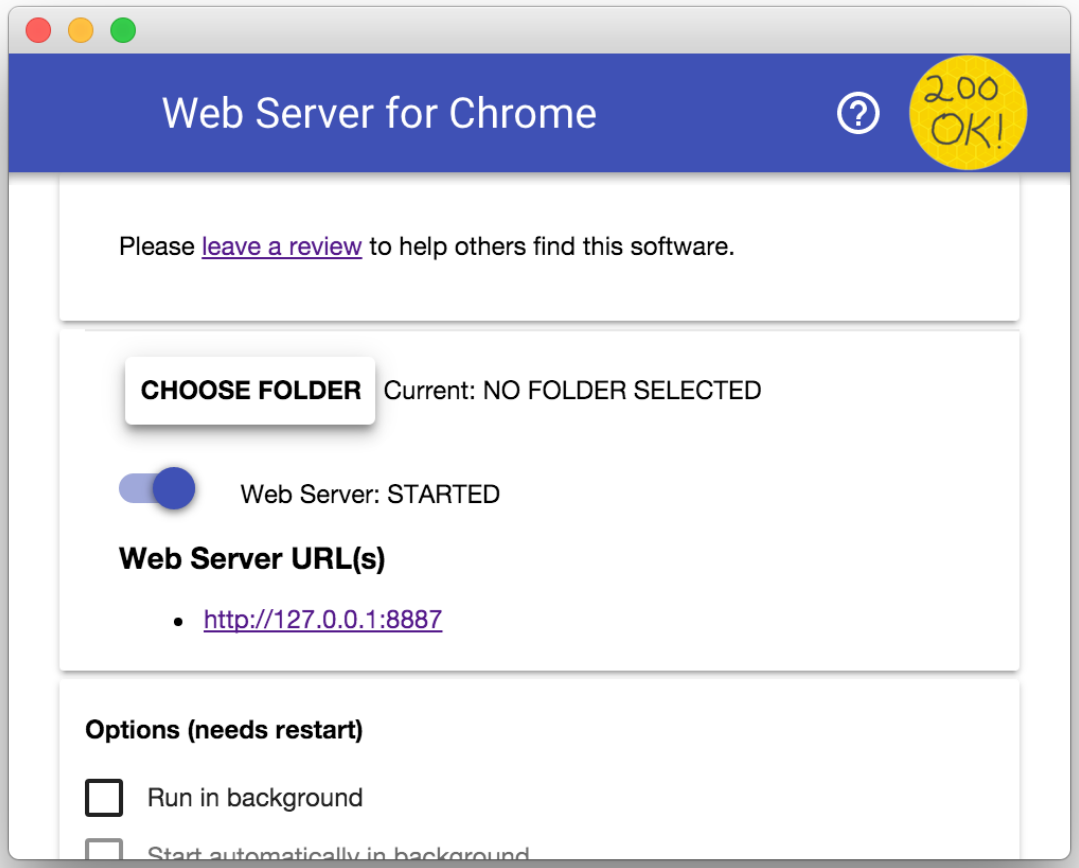
Web Server for Chrome アプリをインストールしたら、ブックマーク バーの [Apps] ショートカットをクリックします。



次のウィンドウで、ウェブサーバー アイコンをクリックします。

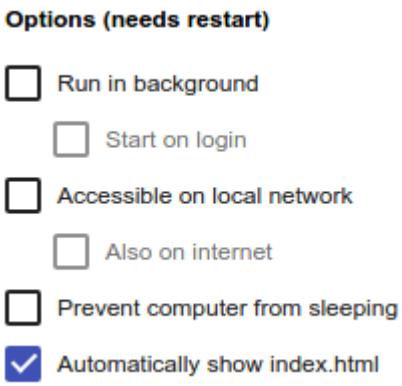


次にこのダイアログが表示され、ローカル ウェブサーバーを構成できます。

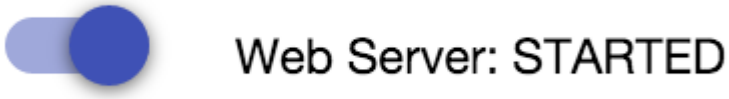


[choose folder] ボタンをクリックし、アプリフォルダを選択します。これにより、ウェブサーバー ダイアログ（**[Web Server URL(s)]** セクション）でハイライト表示された URL から進行中の作業を表示できます。

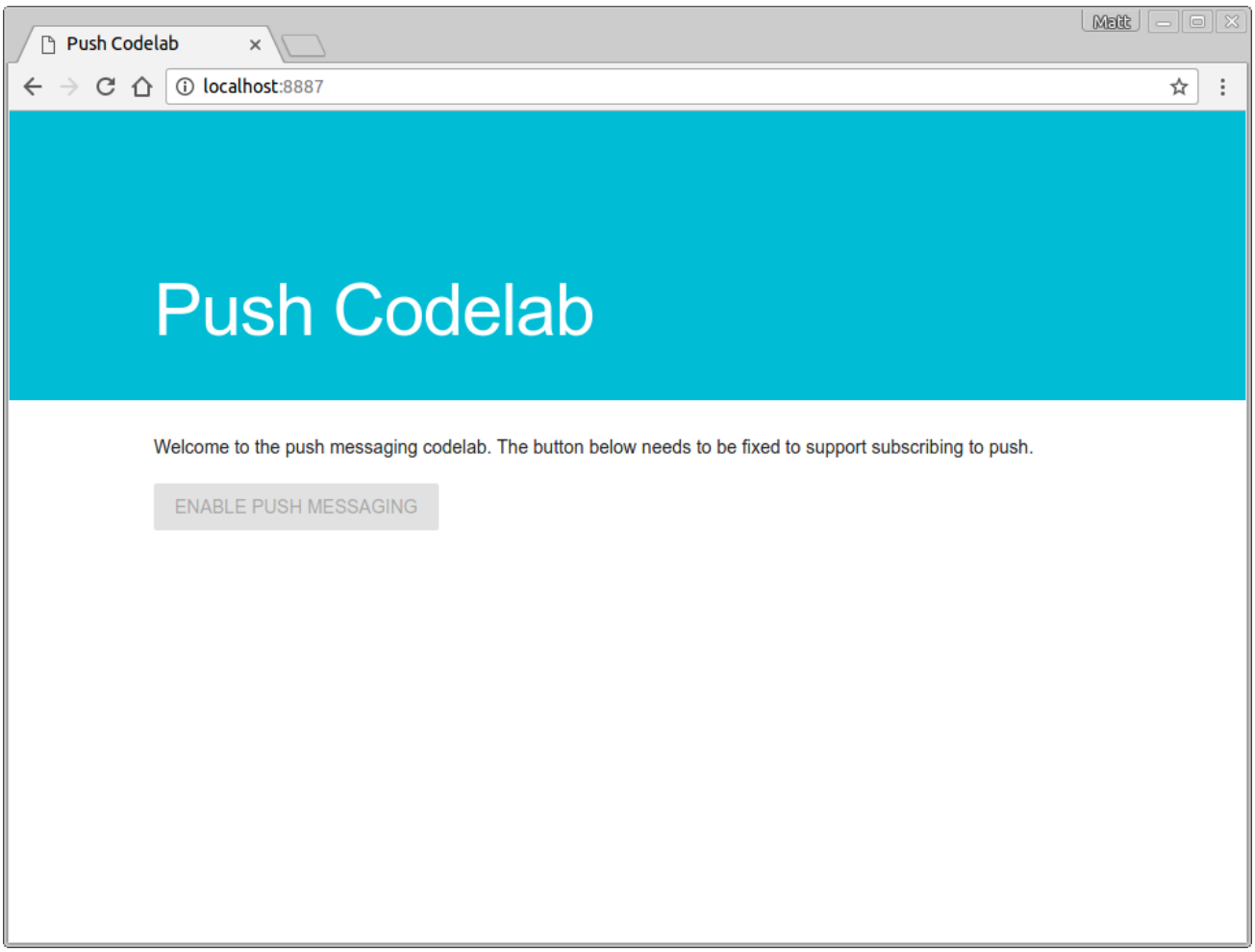
オプションで、以下に示すように [Automatically show index.html] の横のボックスをオンにします。



次に、[Web Server: STARTED] というトグルを左右にスライドしてサーバーを停止したり再開したりします。



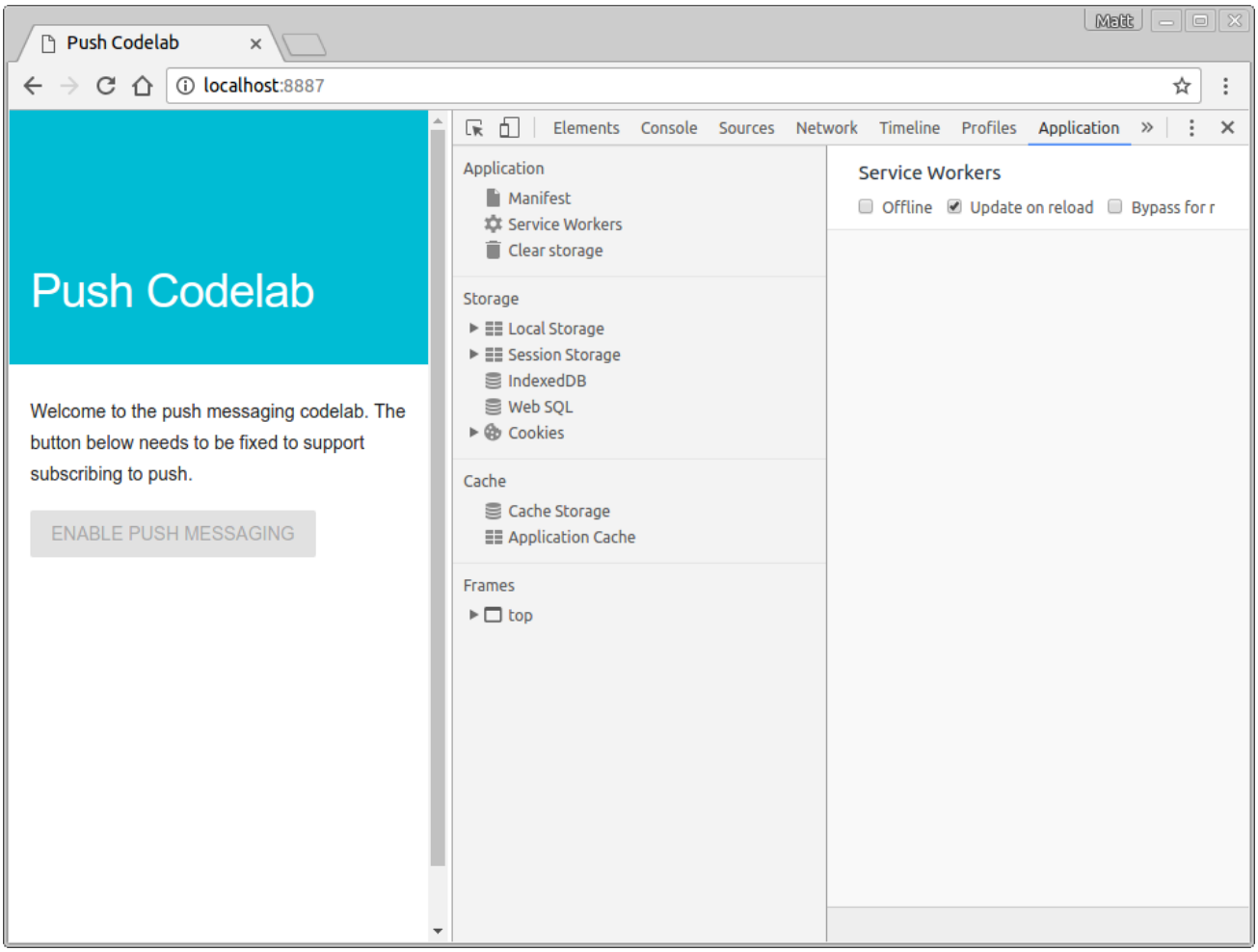
ここで、ご利用のウェブブラウザで（ハイライト表示されたウェブサーバー URL をクリックして）サイトにアクセスすると、次のようなページが表示されます。



常に Service Worker をアップデートする

開発中に Service Worker が常に最新の状態で、最新の変更が反映されているようにすると有用です。

Chrome でこの設定を行うには、DevTools（右クリックして [Inspect]）を開いて **[Application]** パネルに移動し、**[Service Workers]** タブをクリックして **[Update on Reload]** チェックボックスをオンにします。このチェックボックスが有効になっている場合は、ページが再読み込みされるたびに Service Worker が強制的にアップデートされます。



Service Worker の登録

app ディレクトリに **sw.js** という空のファイルがあることに注目してください。これは Service Worker になるファイルですが、現在は空の状態です。後でコードを追加します。

まず、このファイルを Service Worker として登録する必要があります。

app/index.html ページは **scripts/main.js** を読み込みます。この JavaScript ファイルに Service Worker を登録します。

scripts/main.js ファイルに次のコードを追加します。

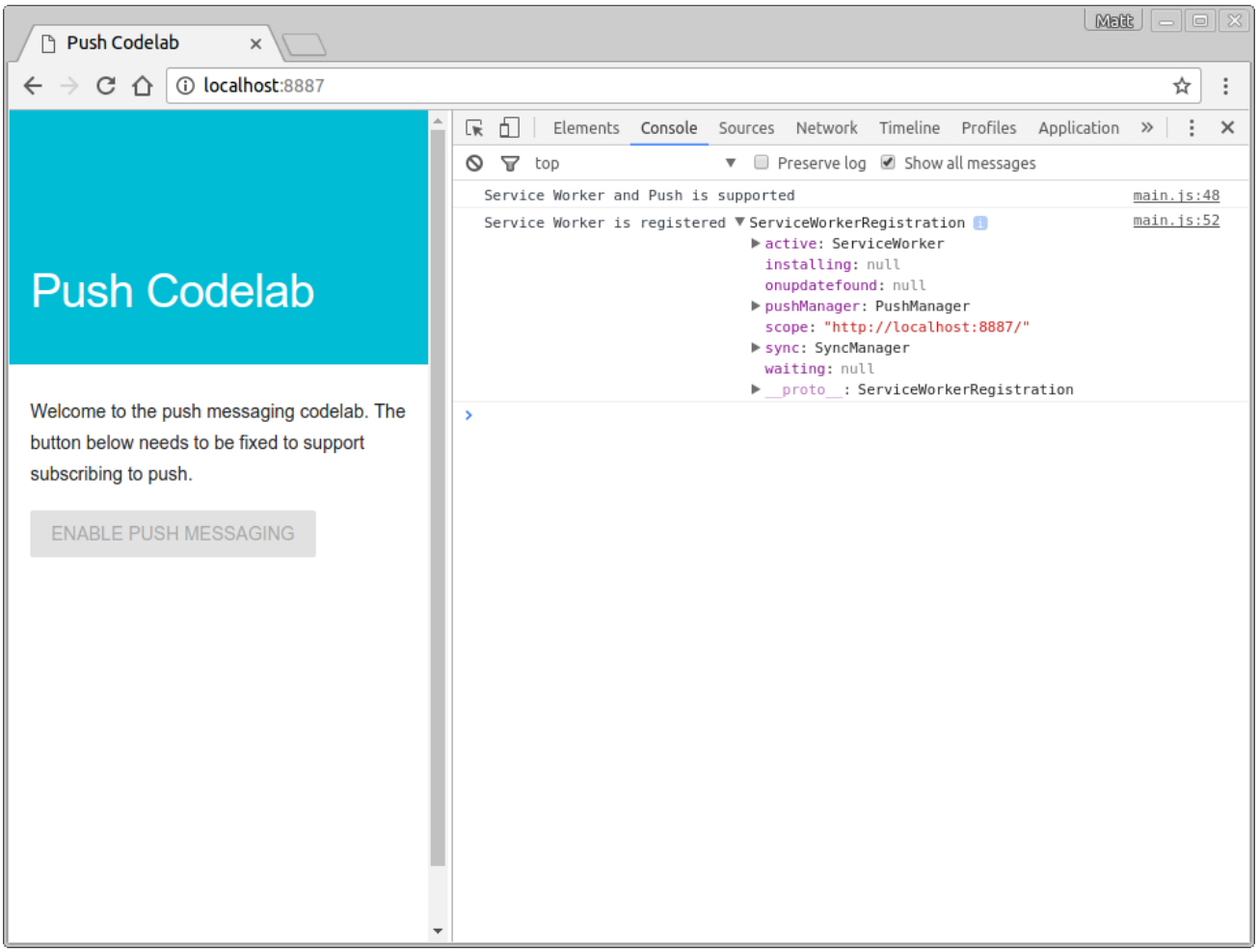
```
if ('serviceWorker' in navigator && 'PushManager' in window) {
  console.log('Service Worker and Push is supported');

  navigator.serviceWorker.register('sw.js')
  .then(function(swReg) {
    console.log('Service Worker is registered', swReg);

    swRegistration = swReg;
  })
  .catch(function(error) {
    console.error('Service Worker Error', error);
  });
} else {
  console.warn('Push messaging is not supported');
  pushButton.textContent = 'Push Not Supported';
}
```

このコードは、Service Worker とプッシュ メッセージが現在のブラウザでサポートされているかどうかを確認し、サポートされている場合は **sw.js** ファイルを登録します。

試してみる

ブラウザで URL **127.0.0.1:8887** を開き、変更を確認します。

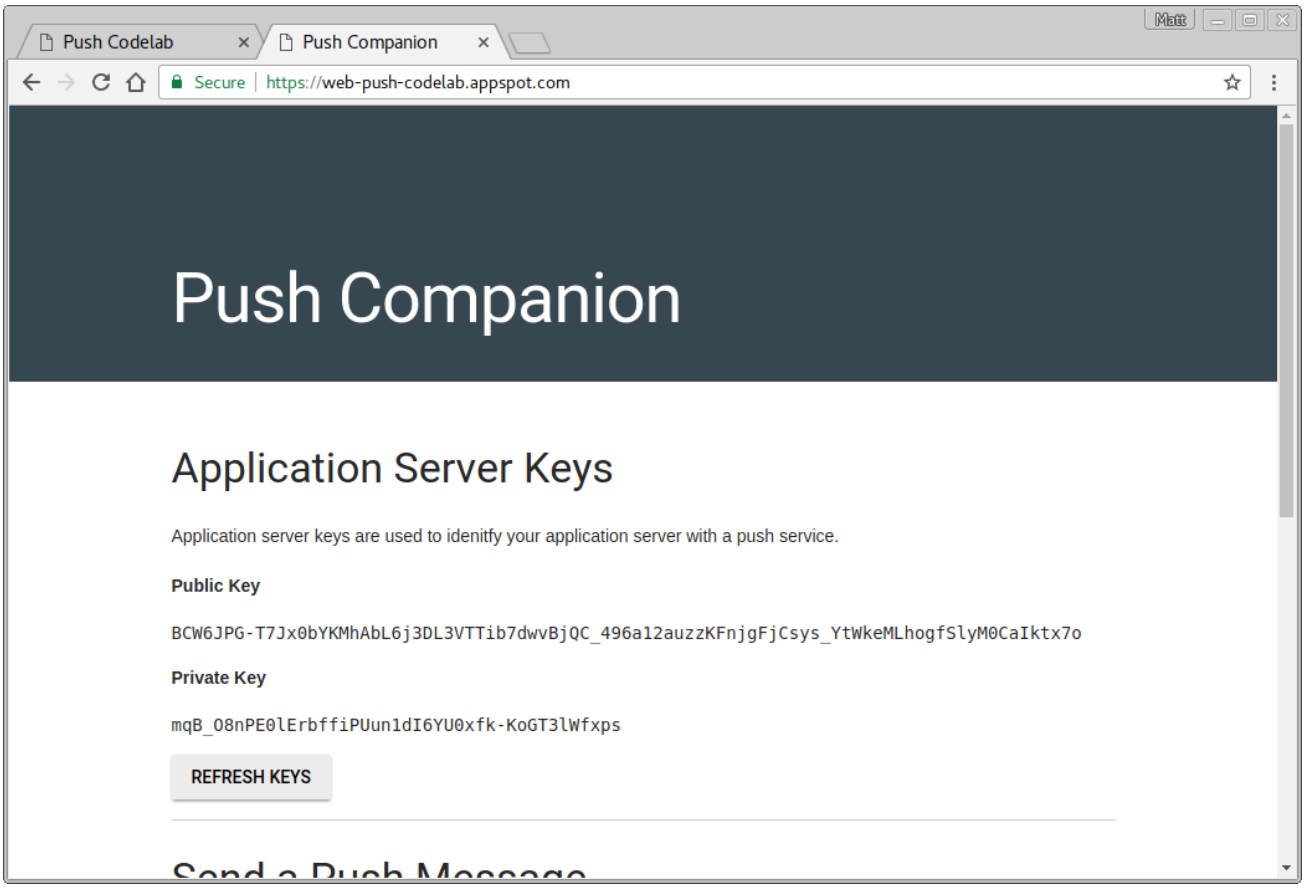
Chrome DevTools を開き、コンソールに **Service Worker is registered** が表示されているか確認します。次のようになります。



アプリケーション サーバーキーの入手

このコードラボでの作業には、アプリケーション サーバーキーをいくつか生成する必要があります。生成するにはこのコンパニオン サイト: https://web-push-codelab.appspot.com/ (https://web-push-codelab.appspot.com/) を使用します。

ここで公開鍵と秘密鍵のペアを生成できます。



公開鍵を **scripts/main.js** にコピーして **<Your Public Key>** の値を置き換えます。

```
const applicationServerPublicKey = '<Your Public Key>';
```

注: 秘密鍵はウェブアプリ内に配置しないでください。

状態の初期化

現在ウェブアプリのボタンは無効になっているため、クリックできません。これは、押しボタンをデフォルトで無効にしておき、プッシュがサポートされて、ユーザーが現在登録されているかどうかを判断できるようになってから有効にすると良いとされているからです。

scripts/main.js で 2 つの関数を作成してみましょう。1 つは **initialiseUI** で、ユーザーが現在登録されているかどうかを確認します。もう 1 つは **updateBtn** で、ボタンを有効にし、ユーザーが登録されているかどうかでテキストを変更します。

initialiseUI 関数は次のようになります。

```
function initialiseUI() {
  // Set the initial subscription value
```

```
  swRegistration.pushManager.getSubscription()
  .then(function(subscription) {
    isSubscribed = !(subscription === null);

    if (isSubscribed) {
      console.log('User IS subscribed.');
    } else {
      console.log('User is NOT subscribed.');
    }

    updateBtn();
  });
}
```

新しいメソッドは、前のステップの **swRegistration** を使用して、その **pushManager** に対して **getSubscription()** を呼び出します。**getSubscription()** は、存在する場合は現在の登録によって解決される Promise を返し、それ以外の場合は **null** を返すメソッドです。これにより、ユーザーが既に登録されているかどうかを確認し、ある状態を設定して **updateBtn()** を呼び出すことができるため、役に立つテキストが表示されたボタンを有効にできます。

次のコードを追加して **updateBtn()** 関数を実装します。

```
function updateBtn() {
  if (isSubscribed) {
    pushButton.textContent = 'Disable Push Messaging';
  } else {
    pushButton.textContent = 'Enable Push Messaging';
  }

  pushButton.disabled = false;
}
```

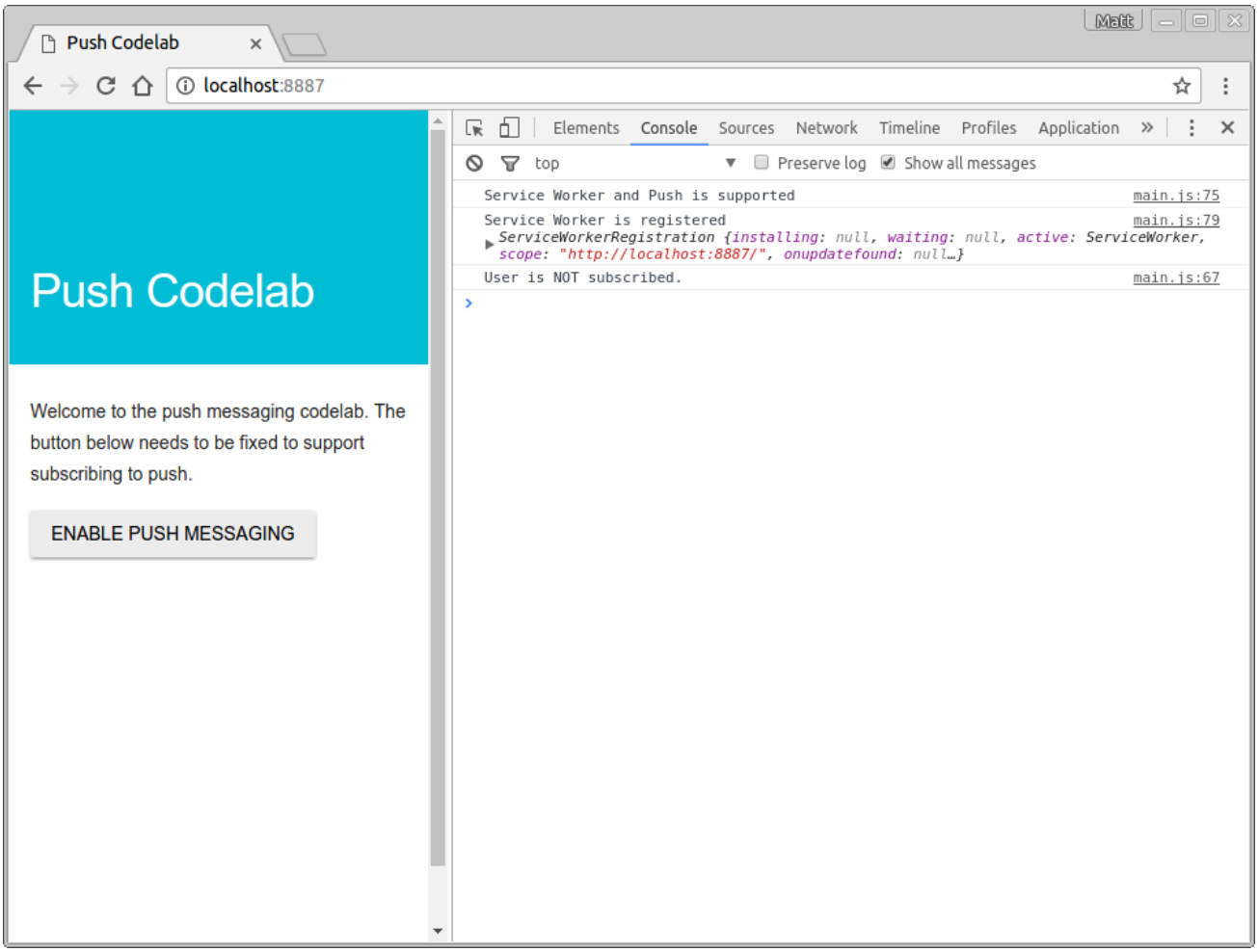
この関数は単純にユーザーが登録されているかどうかに応じてテキストを変更し、ボタンを有効にします。

最後に行うことは、Service Worker が登録されている場合に **initialiseUI()** を呼び出すことです。

```
navigator.serviceWorker.register('sw.js')
.then(function(swReg) {
  console.log('Service Worker is registered', swReg);

  swRegistration = swReg;
  initialiseUI();
})
```

試してみる

ウェブアプリを開くと、[Enable Push Messaging] ボタンが有効（クリックできる）になっており、コンソールに「User is NOT subscribed」が表示されています。



コードラボの残りの部分に進んでいくと、ユーザーが登録や登録解除を行った場合にボタンのテキストが変わるようになります。

ユーザーの登録

現在 [Enable Push Messaging] ボタンは操作できません。これを修正してみましょう。

以下のように **`initialiseUI()`** 関数でボタンにクリック リスナーを追加します。

```
function initialiseUI() {
  pushButton.addEventListener('click', function() {
    pushButton.disabled = true;
    if (isSubscribed) {
```

```
      // TODO: Unsubscribe user
    } else {
      subscribeUser();
    }
  });

  // Set the initial subscription value
  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();
  });
}
```

ユーザーが押しボタンをクリックする際、プッシュの登録に時間がかかっているためにユーザーが再度クリックすることのないように、まず、ボタンを無効にします。

次に、ユーザーが現在登録されていないことがわかると **subscribeUser()** を呼び出し、次のコードをコピーして **scripts/main.js** に貼り付けます。

```
function subscribeUser() {
  const 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) {
    console.log('Failed to subscribe the user: ', err);
    updateBtn();
```

```
  });
}
```

このコードにより実行される内容や、プッシュ メッセージ用にユーザーを登録する方法を、段階を追って見ていきましょう。

最初に、アプリケーション サーバーの公開鍵（base 64 の安全な URL エンコード）を取得し、**UInt8Array** に変換します。これが subscribe 呼び出しの想定される入力です。**scripts/main.js** の上部に既に **urlB64ToUint8Array** 関数があります。

値を変換したら、Service Worker の **pushManager** で **subscribe()** メソッドを呼び出し、アプリケーション サーバーの公開鍵と値 **userVisibleOnly: true** を渡します。

```
const applicationServerKey = urlB64ToUint8Array(applicationServerPublicKey);
swRegistration.pushManager.subscribe({
  userVisibleOnly: true,
  applicationServerKey: applicationServerKey
})
```

userVisibleOnly パラメータは、基本的に、プッシュが送信されるたびに通知を表示するというアドミッションです。執筆時点でこの値は必須で、true にする必要があります。

subscribe() を呼び出すと Promise が返され、この Promise は次のステップ後に解決されます。

1. ユーザーが通知を表示するためのパーミッションを付与します。
2. ブラウザがプッシュ サービスにネットワーク リクエストを送信し、PushSubscription を生成するための詳細を取得します。

これらのステップが成功すると、**subscribe()** Promise は **PushSubscription** で解決されます。ユーザーがパーミッションを付与しない場合やユーザーの登録に問題が発生した場合、Promise はエラーで棄却されます。これにより、コードラボの Promise チェーンは次のようになります。

```
swRegistration.pushManager.subscribe({
  userVisibleOnly: true,
  applicationServerKey: applicationServerKey
})
.then(function(subscription) {
  console.log('User is subscribed:', subscription);

  updateSubscriptionOnServer(subscription);

  isSubscribed = true;
```

```
  updateBtn();

})
.catch(function(err) {
  console.log('Failed to subscribe the user: ', err);
  updateBtn();
});
```

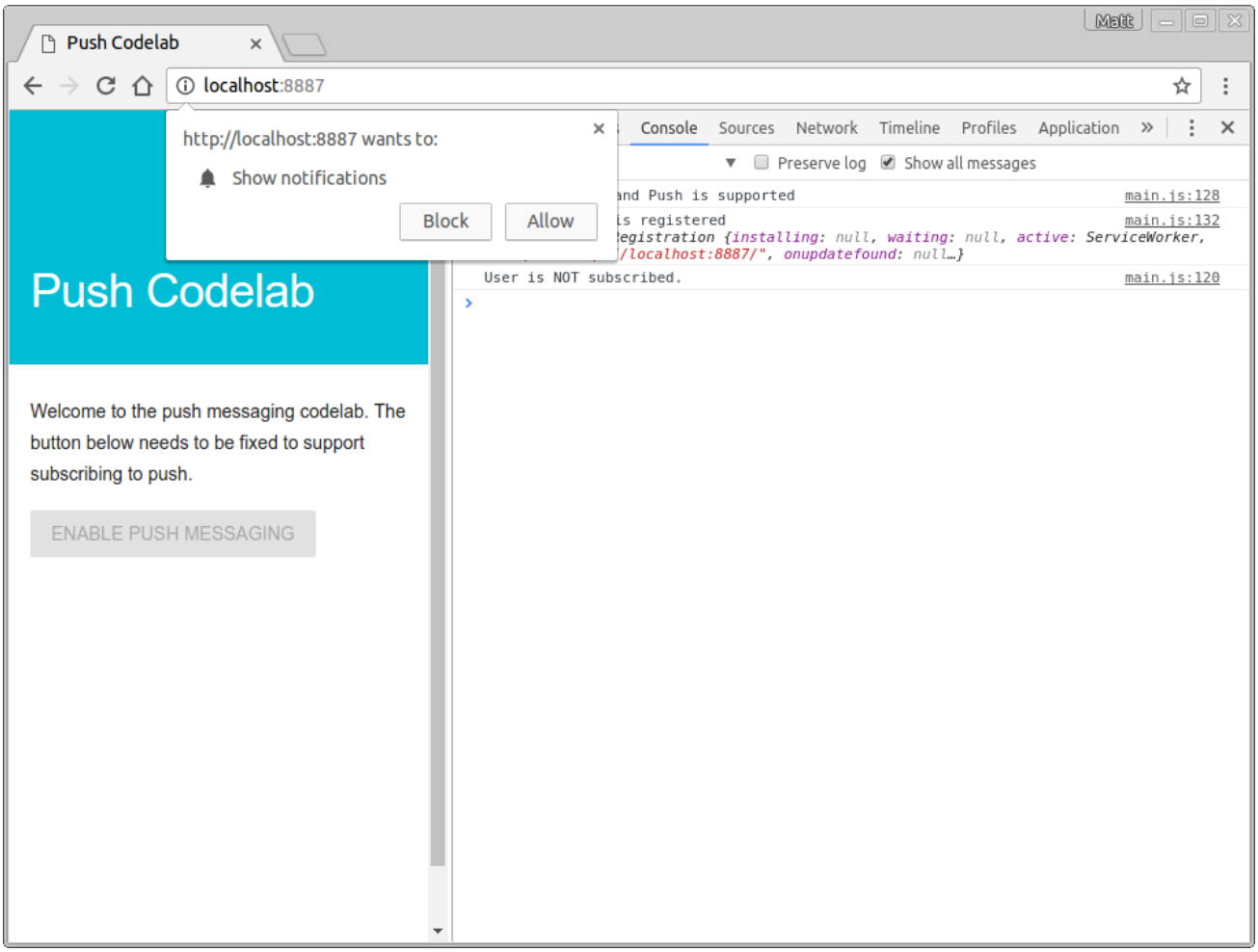
この場合、登録を取得してユーザーを登録済みとして処理するか、エラーを捕捉してコンソールに出力します。どちらの場合も **updateBtn()** を呼び出して、ボタンが再度有効になり、適切なテキストが表示されるようにします。

updateSubscriptionOnServer メソッドは、実際のアプリではバックエンドに登録を送信するメソッドですが、コードラボでは後で役に立つように UI に登録を出力します。このメソッドを **scripts/main.js** に追加します。

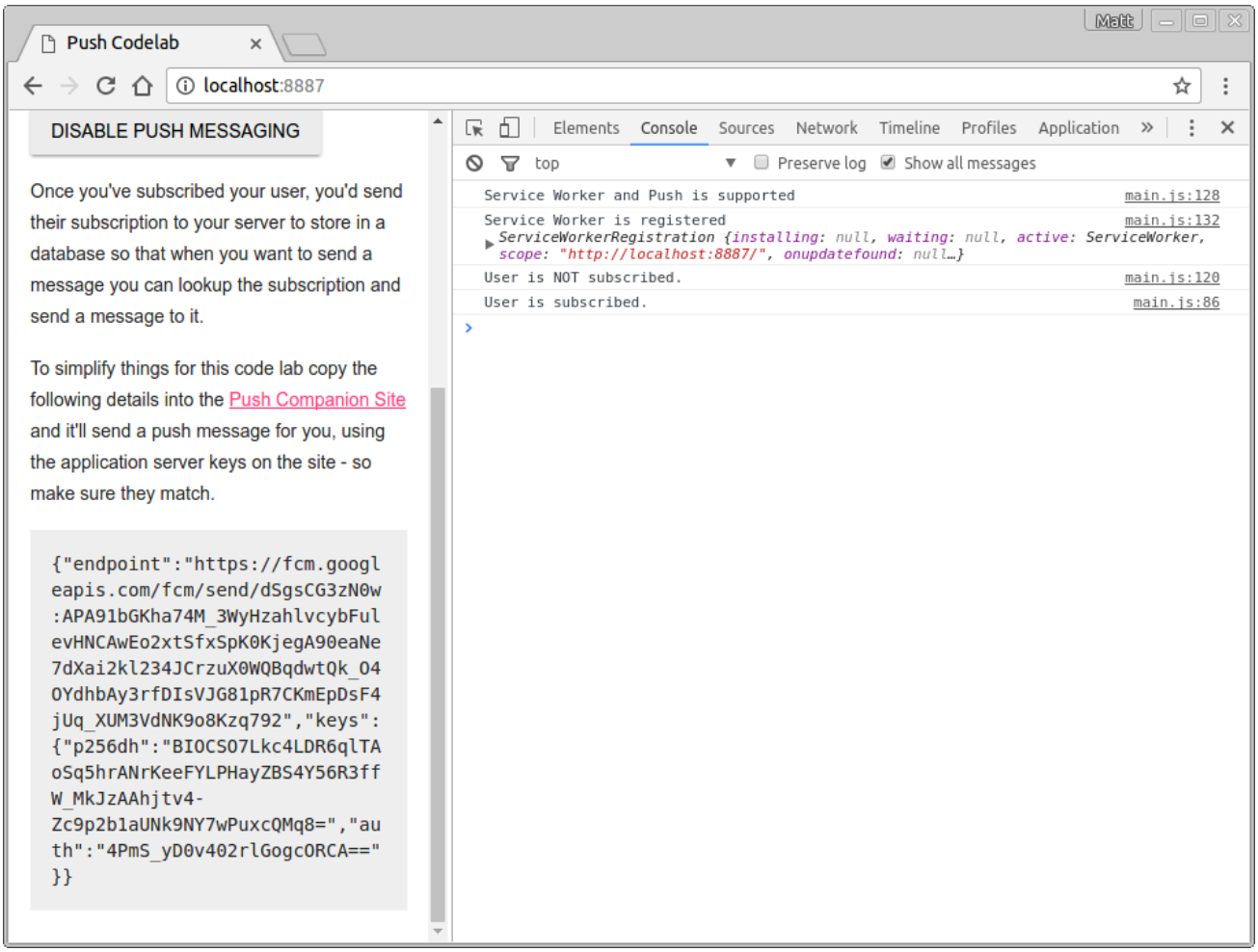
```
function updateSubscriptionOnServer(subscription) {
  // TODO: Send subscription to application server

  const subscriptionJson = document.querySelector('.js-subscription-json');
  const subscriptionDetails =
    document.querySelector('.js-subscription-details');

  if (subscription) {
    subscriptionJson.textContent = JSON.stringify(subscription);
    subscriptionDetails.classList.remove('is-invisible');
  } else {
    subscriptionDetails.classList.add('is-invisible');
  }
}
```

試してみる

ウェブアプリに戻ってボタンをクリックすると、次のようなパーミッション プロンプトが表示されます。



パーミッションを付与すると、コンソールに **`User is subscribed:`** と **`PushSubscription`** の出力が表示され、ボタンのテキストが [Disable Push Messaging] に変わり、ページの下部で JSON 形式の登録を確認できるようになります。



拒否されたパーミッションの処理

まだ対処していないことの 1 つに、ユーザーがパーミッション リクエストをブロックした場合にどうなるかということがあります。この場合、いくつかの特別な配慮が必要です。ユーザーがパーミッションをブロックした場合、ウェブアプリはパーミッション プロンプトを再表示できず、ユーザーを登録できなくなるからです。したがって、少なくとも押しボタンを無効にし、使用不可であることをユーザーが認識できるようにする必要があります。

このシナリオに対処できる場所は **`updateBtn()`** 関数です。次のように **`Notification.permission`** の値を確認するだけです。

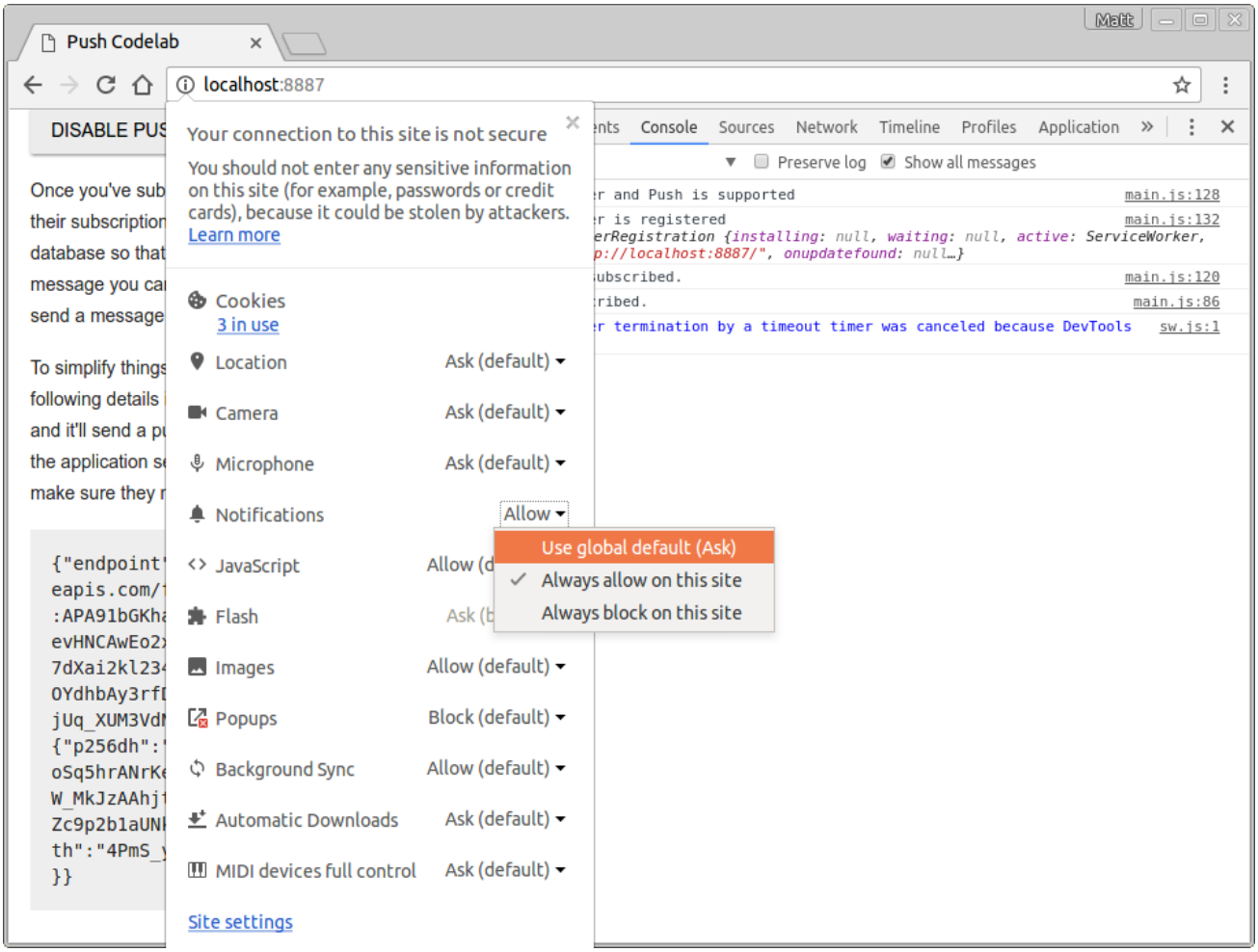
```
function updateBtn() {
  if (Notification.permission === 'denied') {
    pushButton.textContent = 'Push Messaging Blocked.';
    pushButton.disabled = true;
    updateSubscriptionOnServer(null);
    return;
  }
```

```
  if (isSubscribed) {
    pushButton.textContent = 'Disable Push Messaging';
  } else {
    pushButton.textContent = 'Enable Push Messaging';
  }

  pushButton.disabled = false;
}
```

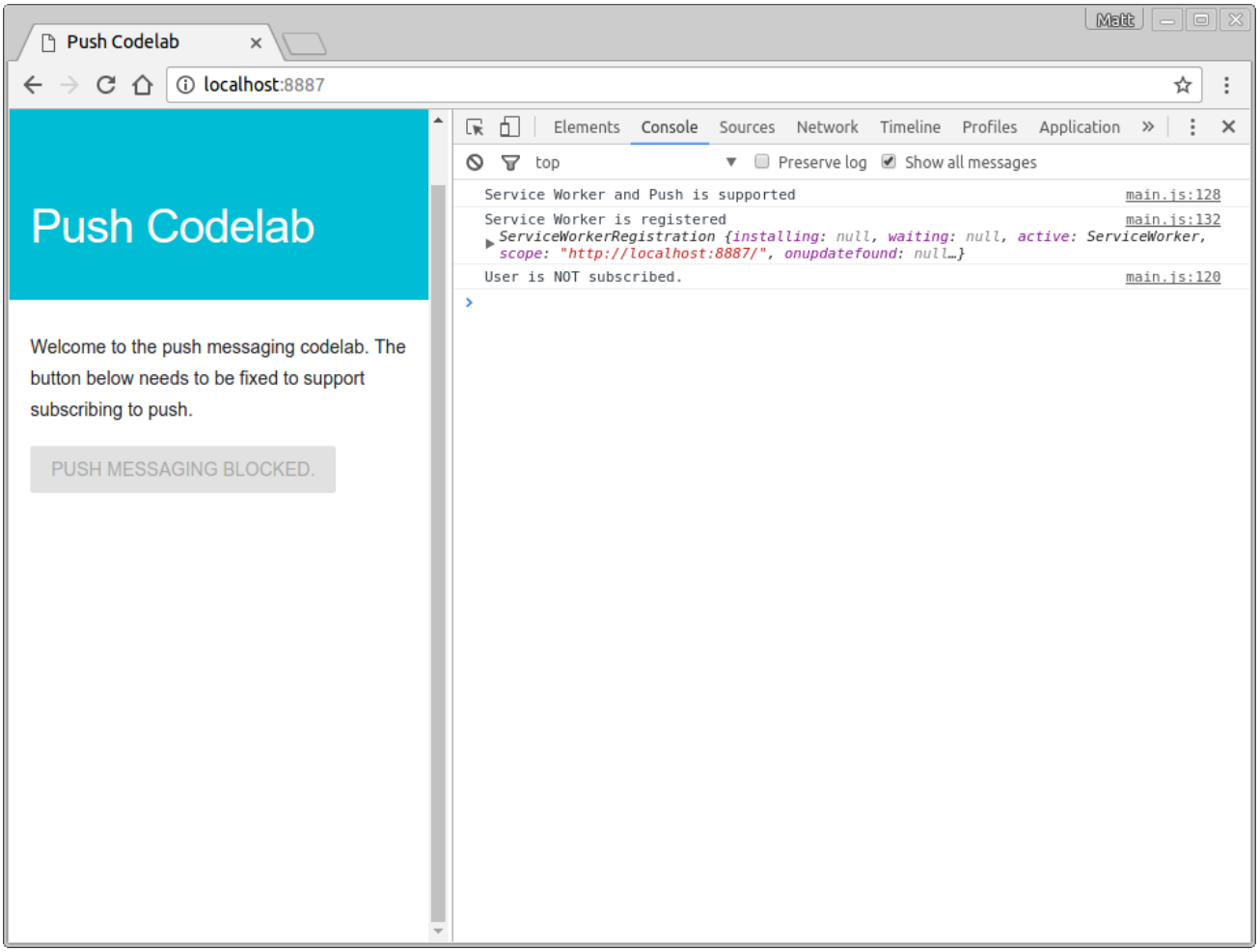
パーミッションが **denied** の場合、ユーザーを登録できず、これ以上何も実行できないため、ボタンを恒久的に無効にすることが最適な方法です。

試してみる

前のステップでウェブアプリのパーミッションを付与したので、URL バーにある円の「i」をクリックして通知パーミッションを [Use global default (Ask)] に変更する必要があります。



この設定を変更したら、ページを更新して [Enable Push Messaging] ボタンをクリックし、パーミッション ダイアログで今回は [Block] を選択します。ボタンのテキストが [Push Messaging Blocked] に変わり、無効になります。



この変更により、ユーザーを登録できるようになり、可能性のあるパーミッション シナリオに対処できます。

プッシュ イベントの処理

バックエンドからプッシュ メッセージを送信する方法を説明する前に、登録されたユーザーがプッシュ メッセージを受信すると実際にどうなるかを考えてみる必要があります。

プッシュ メッセージがトリガーされると、ブラウザはプッシュ メッセージを受信し、このプッシュがどの Service Worker に対するものなのかを判断して、Service Worker を起動してプッシュ イベントを送信します。このイベントをリッスンし、結果として通知を表示する必要があります。

`sw.js` ファイルに次のコードを追加します。

```
self.addEventListener('push', function(event) {
  console.log('[Service Worker] Push Received.');
  console.log(`[Service Worker] Push had this data: "${event.data.text()}"`);

  const title = 'Push Codelab';
  const options = {
    body: 'Yay it works.',
    icon: 'images/icon.png',
    badge: 'images/badge.png'
  };

  event.waitUntil(self.registration.showNotification(title, options));
});
```

このコードを段階を追って見ていきましょう。Service Worker にイベント リスナーを追加することにより、Service Worker でプッシュ イベントをリッスンしています。次のコードです。

```
self.addEventListener('push', ..... );
```

Web Worker を利用したことがない場合、**self** はおそらく初めてでしょう。**self** は Service Worker 自体を参照するため、イベント リスナーを Service Worker に追加しています。

プッシュ メッセージを受信すると、イベント リスナーが起動され、registration で **showNotification()** を呼び出して通知を作成します。**showNotification()** は **title** を必要とし、**options** オブジェクトも指定できます。ここでオプションの本体のメッセージ、アイコン、バッジを設定します（執筆時点では、バッジは Android でのみ使用されます）。

```
const title = 'Push Codelab';
const options = {
  body: 'Yay it works.',
  icon: 'images/icon.png',
  badge: 'images/badge.png'
};
self.registration.showNotification(title, options);
```

プッシュ イベントで最後に説明するのは、**event.waitUntil()** についてです。このメソッドは Promise を取り、ブラウザは Service Worker を稼働状態に保ち、渡された Promise が解決されるまで実行を継続します。

上記のコードをもう少しわかりやすくするには、次のように書き換えることができます。

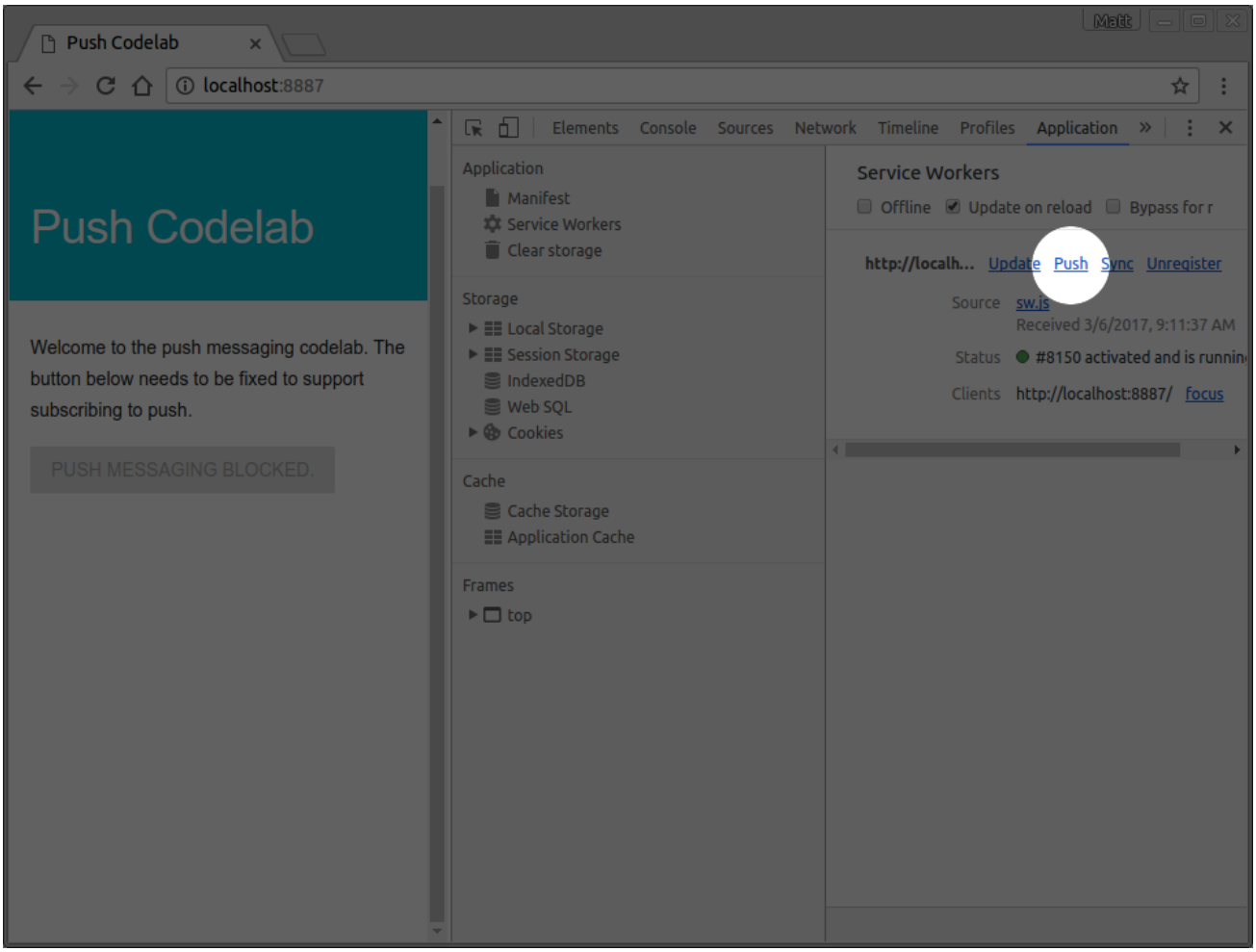
```
const notificationPromise = self.registration.showNotification(title, options
event.waitUntil(notificationPromise);
```

プッシュ イベントを段階を追って見てきました。それでは、プッシュ イベントをテストしてみましょう。

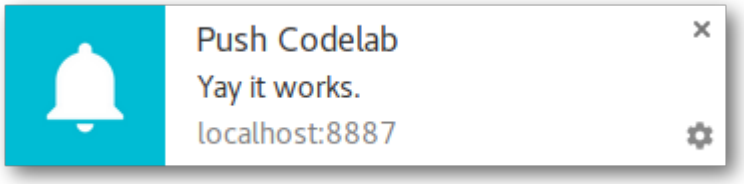
試してみる

Service Worker でプッシュ イベントを使用して、メッセージを受信したときにどうなるかをテストできます。DevTools を使用して疑似プッシュ イベントをトリガーします。

ウェブアプリでプッシュ メッセージを登録し、コンソールに [User IS subscribed] が表示されることを確認してから DevTools の [Application] パネルに移動し、[Service Workers] タブの Service Worker の下の [Push] リンクをクリックします。



クリックすると、次のような通知が表示されます。



注: このステップがうまくいかない場合は、DevTools の [Application] パネルの [Unregister] リンクから Service Worker の登録を解除し、Service Worker が停止するのを待機してからページを再読み込みしてください。

通知のクリック

通知のいずれかをクリックしても何も起こらないことに気付くでしょう。通知のクリックを処理するには、Service Worker で **notificationclick** イベントをリッスンします。

まず、次のように **sw.js** に **notificationclick** リスナーを追加します。

```
self.addEventListener('notificationclick', function(event) {
  console.log('[Service Worker] Notification click Received.');

  event.notification.close();

  event.waitUntil(
    clients.openWindow('https://developers.google.com/web/ (https://developers.googl
  );
});
```

ユーザーが通知をクリックすると、**notificationclick** イベント リスナーが呼び出されます。

このコードラボでは、次のようにして、まず、クリックされた通知を閉じます。

```
event.notification.close();
```

次に URL https://developers.google.com/web/（これは自由に変更できます）を読み込む新しいウィンドウまたはタブを開きます。
(https://developers.google.com/web/（これは自由に変更できます）を読み込む新しいウィンドウまたはタブを開きます。)

```
clients.openWindow('https://developers.google.com/web/ (https://developers.google.com
```

再度 `event.waitUntil()` を呼び出して、新しいウィンドウが表示される前にブラウザが Service Worker を終了しないようにします。

試してみる

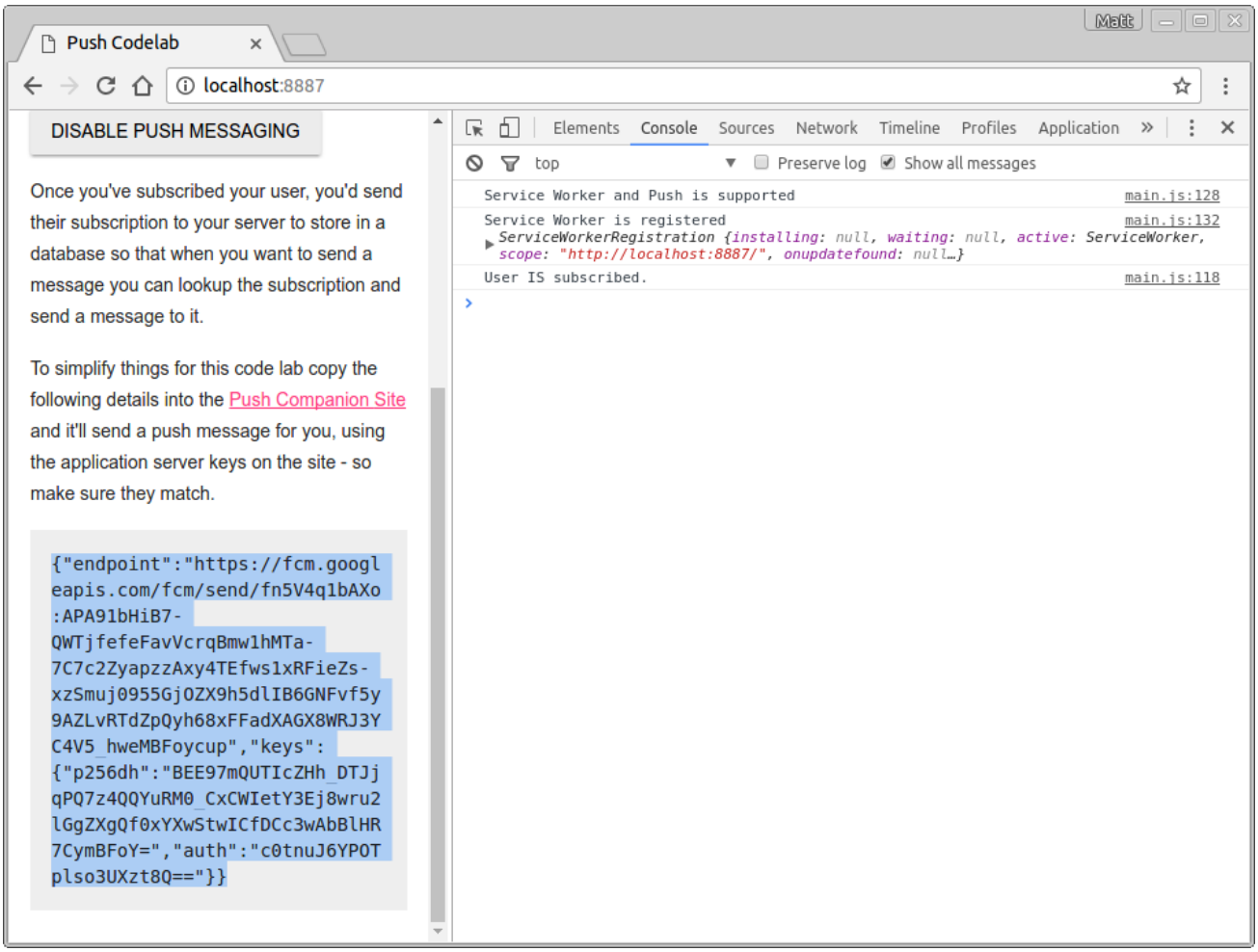
DevTools で再度プッシュ メッセージをトリガーして、通知をクリックします。通知が閉じられ、新しいタブが開きました。

プッシュ メッセージの送信

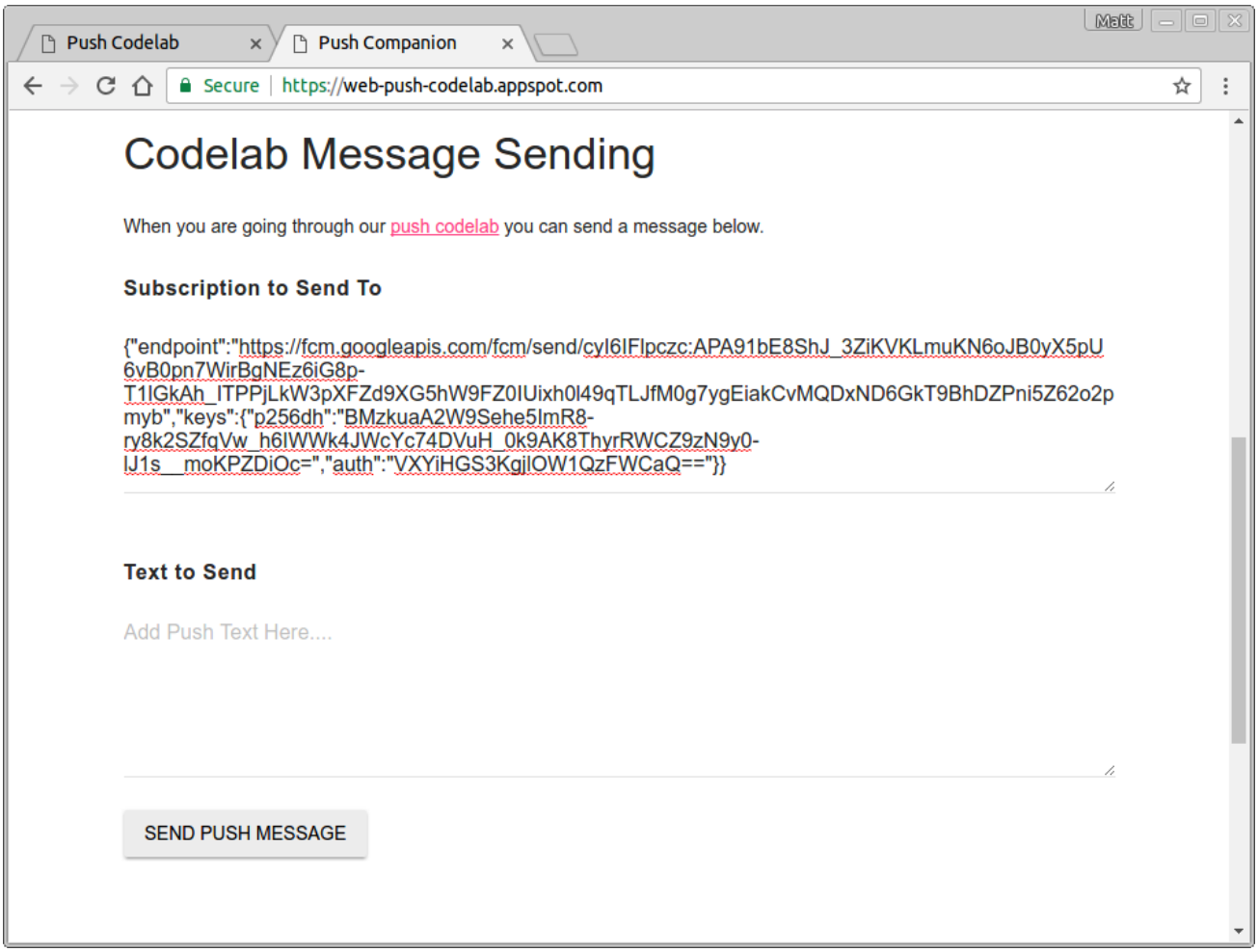
DevTools を使用してウェブアプリで通知を表示できることを確認し、クリックされた通知がどのように閉じるかがわかりました。次のステップでは実際のプッシュ メッセージを送信します。

通常このプロセスはウェブページからバックエンドに登録を送信し、バックエンドは登録のエンドポイントに対する API 呼び出しを生成してプッシュ メッセージをトリガーします。

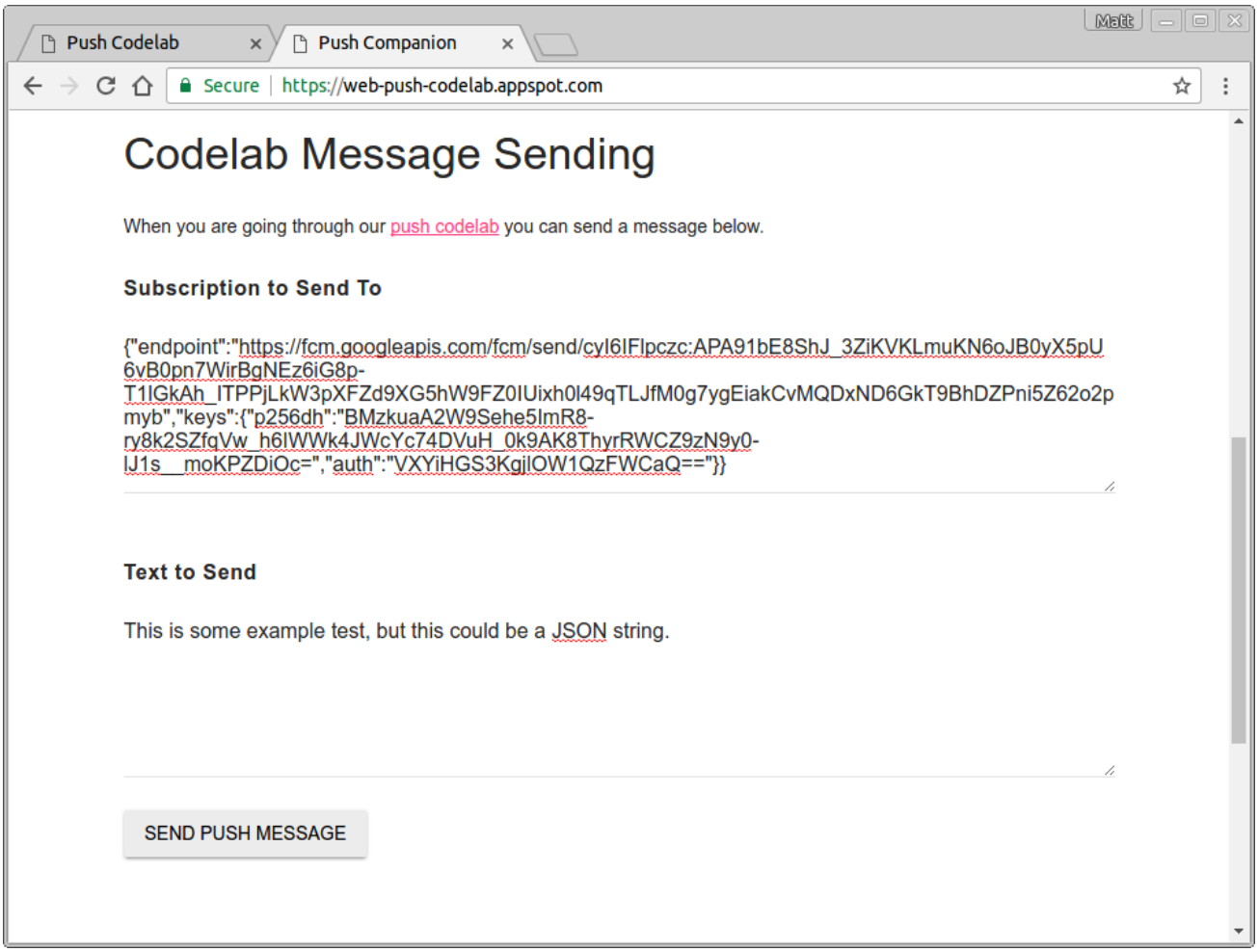
これはこのコードラボの範囲外ですが、このコードラボのコンパニオン サイト（https://web-push-codelab.appspot.com/ (https://web-push-codelab.appspot.com/)）を使用して実際のプッシュ メッセージをトリガーできます。ページの下部の登録をコピーして貼り付けます。



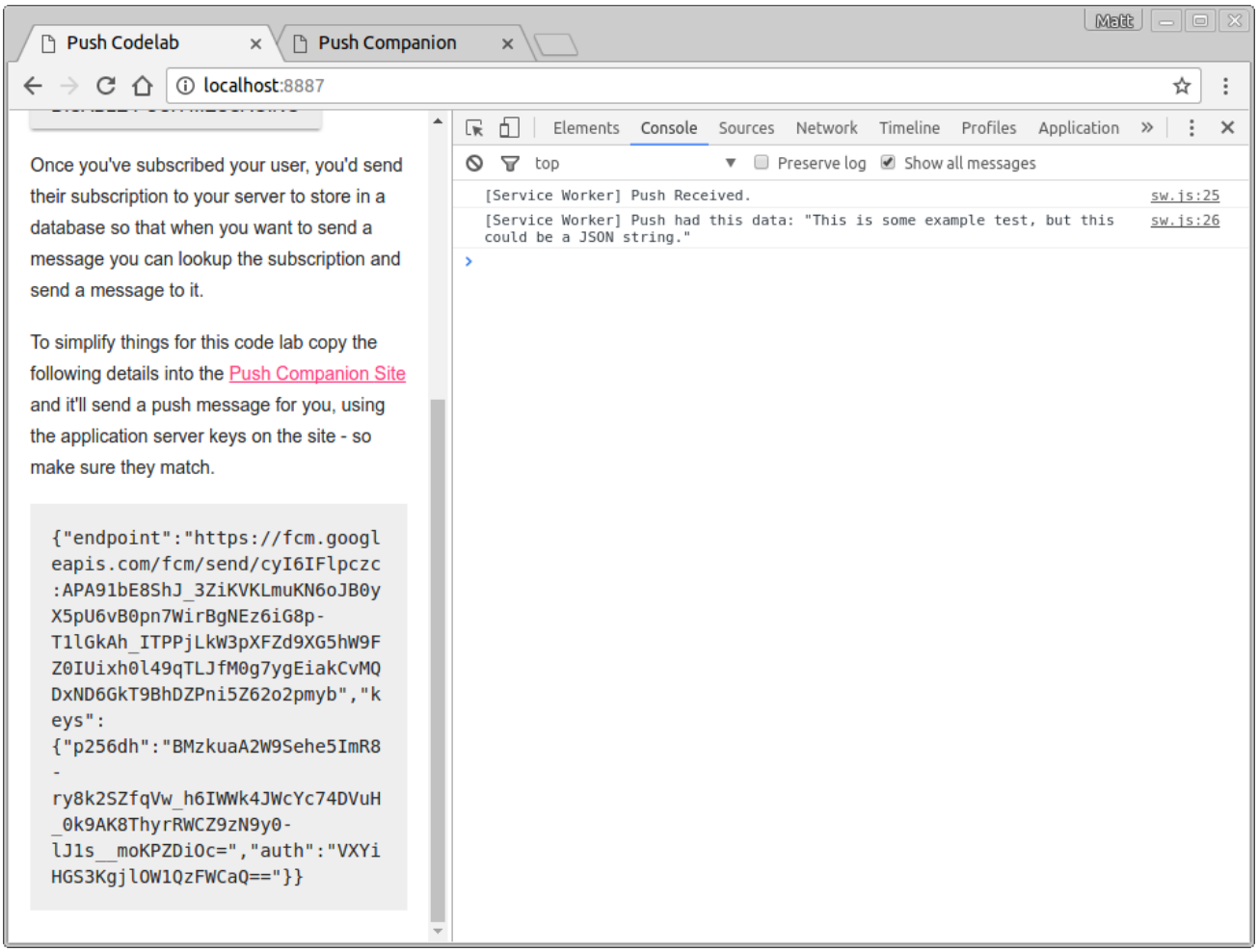
次にコンパニオン サイトの [Subscription to Send To] テキスト領域にこれを貼り付けます。



次に [Text to Send] にプッシュ メッセージで送信する文字列を入力し、最後に [Send Push Message] ボタンをクリックします。



そうするとプッシュ メッセージを受信し、入力したテキストがコンソールに出力されます。



これにより、データの送受信をテストして、結果として通知を操作できます。

このコンパニオン アプリは、web-push library (https://github.com/web-push-libs/web-push) を使用してメッセージを送信している実際のノード サーバーです。プッシュ メッセージの送信に利用できるライブラリを確認するために web-push-libs org on Github (https://github.com/web-push-libs/) を確認してみる価値はあります（これはプッシュ メッセージをトリガーするために多数の具体的な詳細を処理します）。

ユーザーの登録の解除

現在プッシュからユーザーの登録を解除できません。これを行うには、**PushSubscription** で **unsubscribe()** を呼び出す必要があります。

scripts/main.js ファイルに戻り、**initialiseUI()** の **pushButton** のクリック リスナーを次のように変更します。

```
pushButton.addEventListener('click', function() {
  pushButton.disabled = true;
  if (isSubscribed) {
```

```
    unsubscribeUser();
  } else {
    subscribeUser();
  }
});
```

新しい関数 **unsubscribeUser()** を呼び出していることに注目してください。このメソッドでは、現在の登録と呼び出される登録の解除を取得します。**scripts/main.js** ファイルに次のコードを追加します。

```
function unsubscribeUser() {
  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();
  });
}
```

この関数を段階を追って見ていきましょう。

まず、**getSubscription()** を呼び出して現在の登録を取得します。

```
swRegistration.pushManager.getSubscription()
```

これにより、登録が存在する場合は **PushSubscription** で解決する Promise が返され、存在しない場合は **null** が返されます。登録が存在する場合は **unsubscribe()** を呼び出し、**PushSubscription** を無効にします。

```
swRegistration.pushManager.getSubscription()
.then(function(subscription) {
  if (subscription) {
    // TODO: Tell application server to delete subscription
    return subscription.unsubscribe();
  }
```

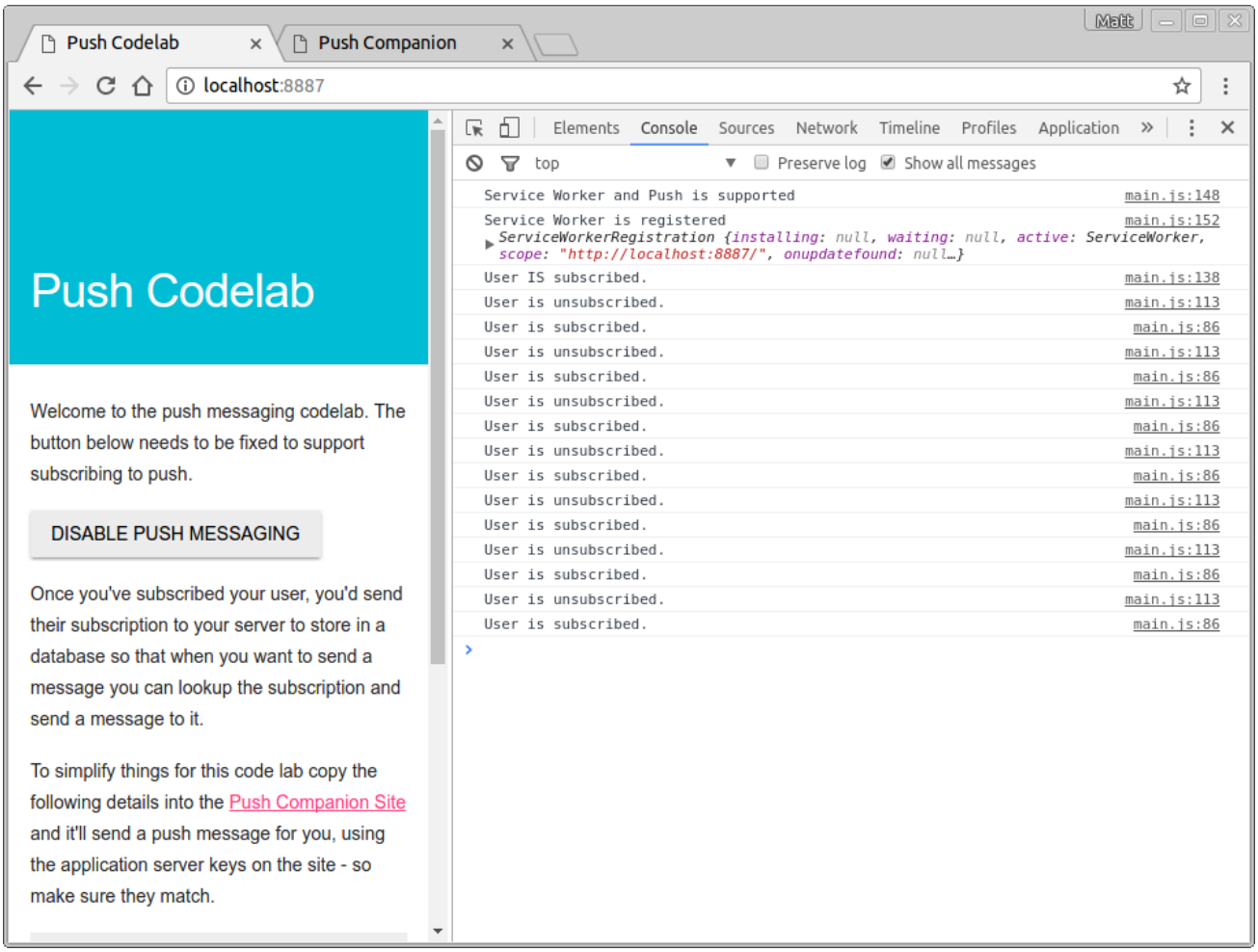
```
})
.catch(function(error) {
  console.log('Error unsubscribing', error);
})
```

unsubscribe() を呼び出すと、完了に時間がかかる可能性がある場合は Promise が返されるのでその Promise を返すと、チェーンの次の **then()** が **unsubscribe()** の完了を待機します。**unsubscribe()** の呼び出しがエラーになる場合に備えて、catch ハンドラも追加します。この後 UI をアップデートできます。

```
.then(function() {
  updateSubscriptionOnServer(null);

  console.log('User is unsubscribed.');
  isSubscribed = false;

  updateBtn();
})
```

試してみる

ウェブアプリで [Enable Push Messaging] または [Disable Push Messaging] を押すことができ、ログにユーザーが登録されていることや登録が解除されていることが表示されます。



完了

これでこのコードラボは完了です。

このコードラボでは、プッシュをウェブアプリに追加して使用できるようにする方法を説明しました。ウェブ通知で実行できる内容の詳細については、これらのドキュメントをご覧ください (https://developers.google.com/web/fundamentals/engage-and-retain/push-notifications/)。

サイトにプッシュをデプロイする場合は、GCM を使用する古い標準に準拠していないブラウザのサポートを追加する必要がある場合があります。詳細については、こちらをご覧ください (https://web-push-book.gauntface.com/chapter-06/01-non-standards-browsers/)。

参考資料

- Web**Fundamentals** のウェブでのプッシュ通知 (https://developers.google.com/web/fundamentals/engage-and-retain/push-notifications/)のドキュメント

- ウェブプッシュ ライブラリ (https://github.com/web-push-libs/) - Node.js、PHP、Java、Python などのウェブプッシュ ライブラリ

関連するブログ投稿

- ウェブプッシュ ペイロードの暗号化 (https://developers.google.com/web/updates/2016/03/web-push-encryption)
- アプリケーション サーバーキーとウェブでのプッシュ (https://developers.google.com/web/updates/2016/07/web-push-interop-wins)
- 通知アクション (https://developers.google.com/web/updates/2016/01/notification-actions)
- アイコン、クローズ イベント、再通知プリファレンス、タイムスタンプ (https://developers.google.com/web/updates/2016/03/notifications)

問題の発見やフィードバック

コードラボの向上のために問題 (https://github.com/googlechrome/push-notifications/issues)がある場合はお知らせください。 ご協力お願いします。

Last updated 5月 18, 2017.

Chromium Blog
The latest news on the Chromium blog

GitHub
Fork our API code samples and other open-source projects.

Twitter
Connect with @ChromiumDev on Twitter

Videos
Check out the Web Developer Relations team's videos

Events
Attend a developer event and get hacking