Android payment app integration into Android web browsers

THIS DOCUMENT IS PUBLIC

This document is now on the Web.Dev, although that has fewer technical details.

Author: rouslan@chromium.org
Last update: 2017.9.27

This is a proposal for integration of native Android payment apps into Android web browsers. The goal of this design is to let *any* native Android payment app work with *any* Android web browser without the need for browsers to link against every payment app SDK in the world.

Background

Web payments is a W3C standard API for e-commerce websites to collect payment information from users with user consent. Native Android payment app support should be added to Android web browsers to let users pay in their preferred way. Web browsers cannot link in every payment app SDK in the world. This document describes a generic method for any payment app to work with any web browser through Android intents.

Requirements

- **Browser permissions**: A payment app needs to know where the money is going. Therefore, a payment app should be able to control which browser is allowed to call it.

- **Website permissions**: A payment app should be able to control which website is allowed to invoke it. Therefore, a browser should pass the origin and certificate of the calling website to the payment app.

- **Payment app permissions**: If a merchant specifies that they accept a certain payment app, then the merchant needs to know that an impersonator app will not be stealing user’s credentials. Therefore, a payment app should be able to control which apps can respond to its payment method identifier. For example, “https://bobpay.com” may be locked down to only Bob Pay native Android app. On the other hand, “http://www.alice.com/web-pay” may allow any payment app to provide payments.

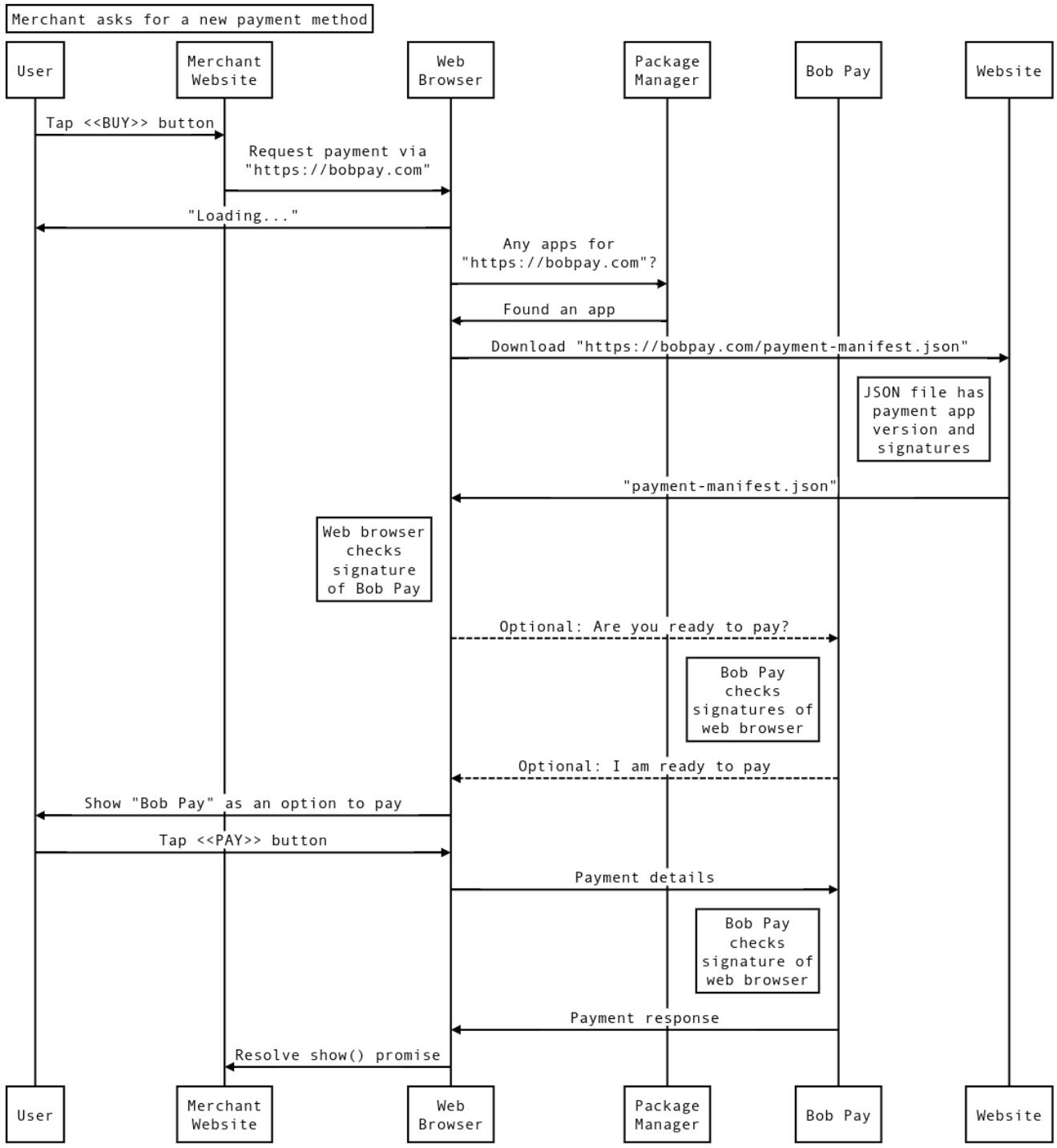
- **Loose integration**: Web browsers should not be compiling against payment app SDKs. Payment apps should not be compiling against web browser SDKs.

- **Performance**: The integration should be fast. The worst case scenario would be a device with 512MB of RAM, cold start for both the browser and the payment app.

- **Install awareness**: If a user installs or uninstalls a payment app, a web browser should be aware of this change. Even if a web browser is installed after a payment app, the browser should be able to use the pre-installed payment app for web payments.

Design

Check if a valid payment app is installed



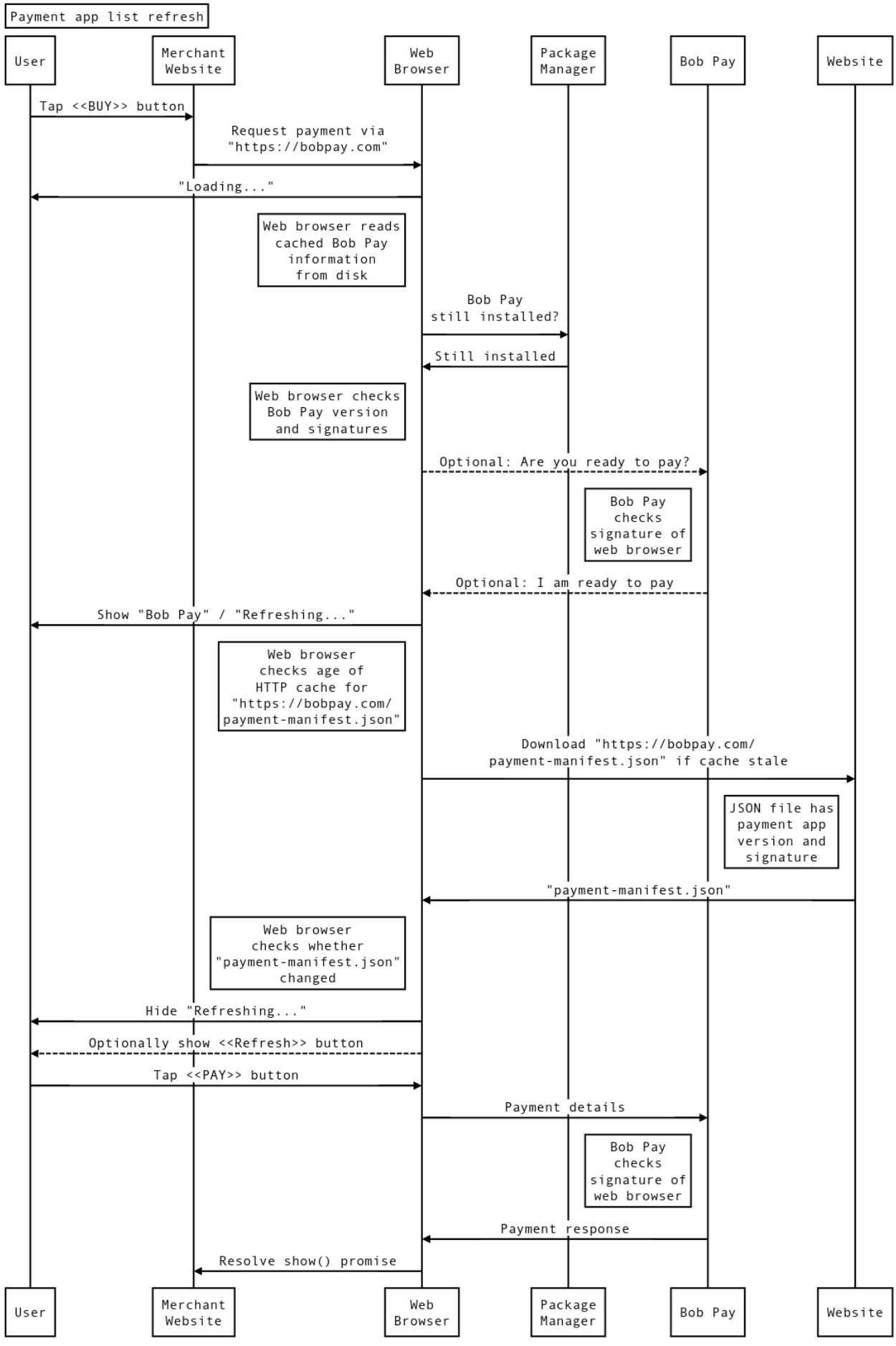
When a merchant requests payment via “https://bobpay.com” method, the web browser queries the Package Manager for any app that can respond to “https://bobpay.com” intent. (Checking locally installed apps first reduces the number of server requests for “payment-manifest.json” file.) If such app is found, then the browser downloads the HEAD of “https://bobpay.com”, and downloads the JSON file pointed to by the HTTP header link with `rel="payment-method-manifest"` attribute. Example of such HTTP header:

```
Link: <payment-manifest.json>; rel="payment-method-manifest"
```

Then the browser downloads “https://bobpay.com/payment-manifest.json”, which contains pointers to the default applications of that payment method. Example of such payment method manifest:

```
{"default_applications": ["https://bobpay.com/bobpay-app.json"]}
```

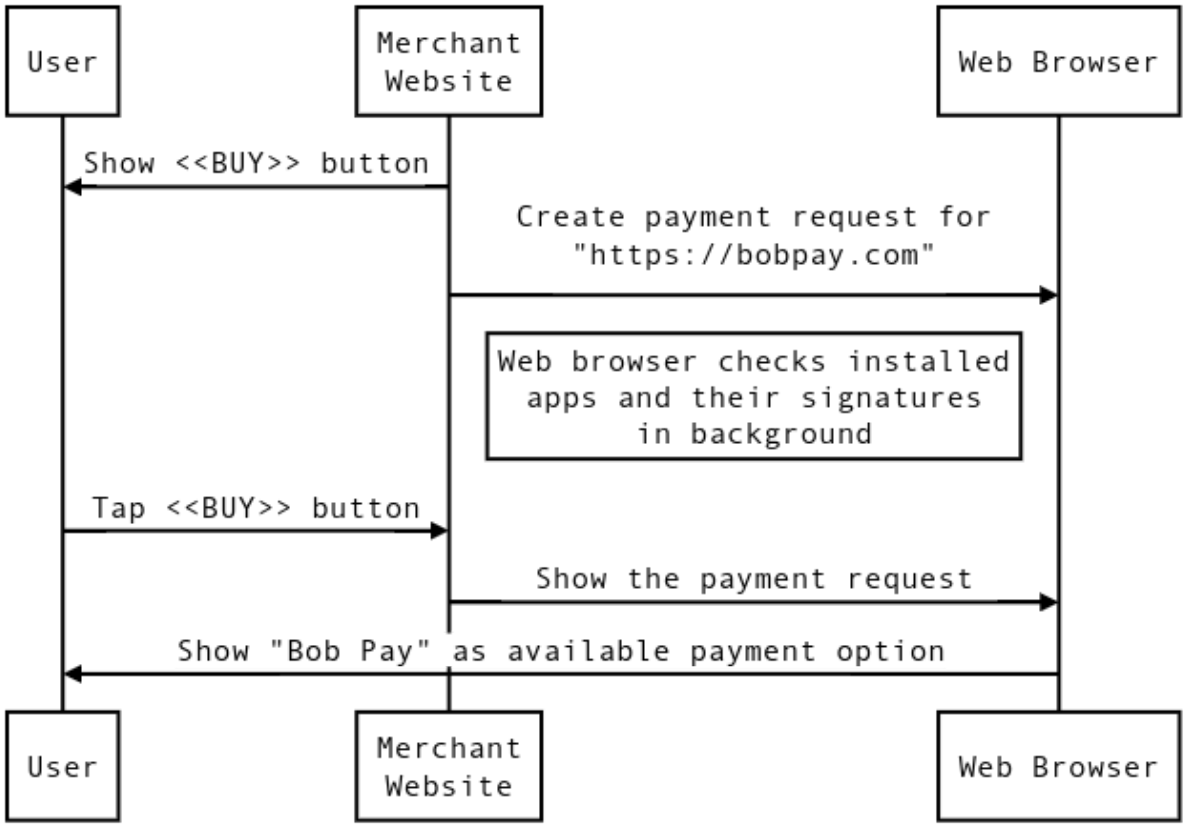
The browser downloads “https://bobpay.com/bobpay-app.json” and verifies the installed app against the version and signatures in “bobpay-app.json”. All downloads must be over HTTPS. HTTP response codes must be 200. HTTP redirects are not followed.


Payment app list refresh
User
Merchant Website
Web Browser
Package Manager
Bob Pay
Website
Tap <<BUY>> button
Request payment via "https://bobpay.com"
"Loading..."
Web browser reads cached Bob Pay information from disk
Bob Pay still installed?
Still installed
Web browser checks Bob Pay version and signatures
Optional: Are you ready to pay?
Bob Pay checks signature of web browser
Optional: I am ready to pay
Show "Bob Pay" / "Refreshing..."
Web browser checks age of HTTP cache for "https://bobpay.com/payment-manifest.json"
Download "https://bobpay.com/payment-manifest.json" if cache stale
JSON file has payment app version and signature
"payment-manifest.json"
Web browser checks whether "payment-manifest.json" changed
Hide "Refreshing..."
Optionally show <<Refresh>> button
Tap <<PAY>> button
Payment details
Bob Pay checks signature of web browser
Payment response
Resolve show() promise
User
Merchant Website
Web Browser
Package Manager
Bob Pay
Website

After the browser has been used for web payments at least once, it has a cache of locally installed payment apps. This cache allows for faster display of payment UI. When user taps on the merchant website <<BUY>> button, the browser immediately shows the cached list of apps with a “Refreshing…” indicator. When the payment app cache has refreshed, the user can tap a <<Refresh>> button to see the updated list of payment apps.

Protecting the cache from malware is out of scope of this project. If the user has phone malware that can read and write other apps' data directories, the malware would be able to read user's credit card numbers, addresses, and passwords from disk, for example. Guarding against malware on OS is orthogonal to this project, but is good to keep in mind. See https://developer.android.com/guide/topics/data/data-storage.html for details.

Preloading

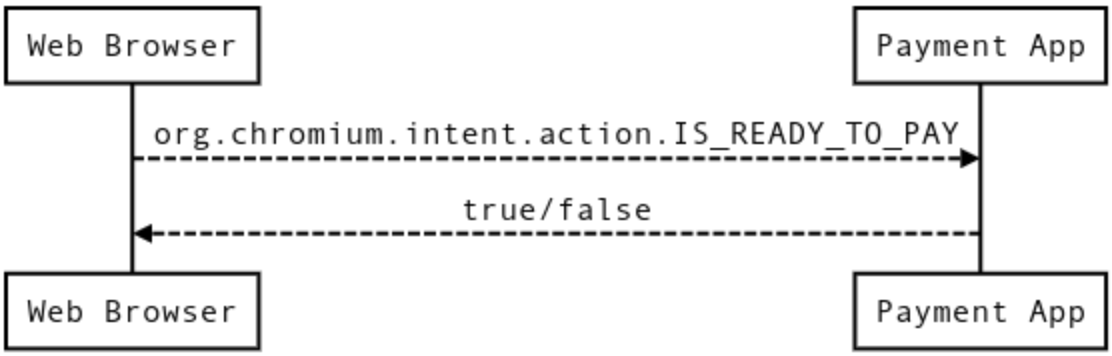


A web browser queries installed payment apps when the JavaScript PaymentRequest object is constructed. A website can create a PaymentRequest object when showing the <<BUY>> button, but call `PaymentRequest.show()` only when user taps this <<BUY>> button. This allows for faster UI response.

Messages

Browsers and payment apps pass data to each other via Intent extras, which are key-value string pairs.

Optional: “Is ready to pay”



If the payment app has a service with `"IS_READY_TO_PAY"` Android intent handler, then the web browser can check with the payment app before showing it as an option for payment.

“Is ready to pay” parameters

- `ArrayList<String> methodNames` - The names of the method being queried. The elements are the keys in `methodData` dictionary.
- `Bundle[String] methodData` - A mapping from each of the `methodName` to the output of `JSON.stringify(methodData[methodName].data)`.
- `String topLevelOrigin` - The scheme-less origin of the top-level browsing context. For example, “https://mystore.com/checkout” will be passed as “mystore.com”.
- `Parcelable[] topLevelCertificateChain` - The certificate chain of the top-level browsing context. Null for localhost and file on disk, which are both secure contexts without SSL certificates. (The certificate chain is necessary because a payment app might have different trust requirements for websites.)
- `String paymentRequestOrigin` - The schemeless origin of the iframe browsing context that invoked `new PaymentRequest(methodData, details, options)` constructor. If the constructor was invoked from top-level context, then the value of this parameter equals the value of `topLevelOrigin` parameter.

Not all browsers have the capability to determine the values for all of these parameters. Therefore, the payment app should check for existence of these parameters before accessing them.

These parameters are sent to the payment app using intent extras.

```
  Bundle extras = new Bundle();
  extras.putString("key", "value");
  intent.putExtras(extras);
```

The certificate chain is serialized as follows.

```
Parcelable[] certificateChain;
Bundle certficate = new Bundle();
certificate.putByteArray("certificate", certificateByteArray[i]);
certificateChain[i] = certificate;
extras.putParcelableArray("certificateChain", certificateChain);
```

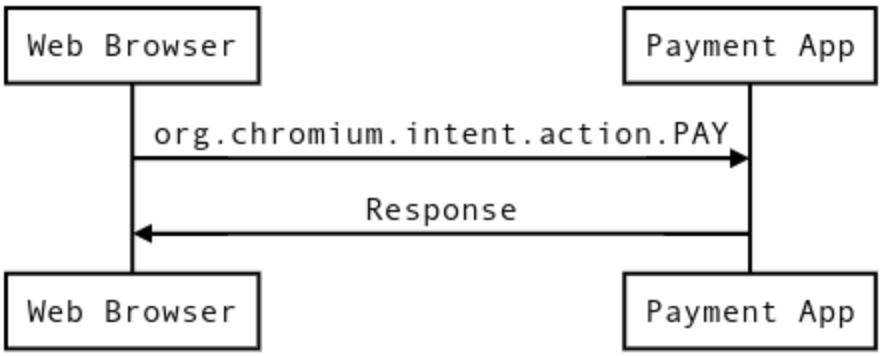
“Is ready to pay” response

- `boolean readyToPay` - Whether the payment app is ready to pay.

The response is sent back via `handleIsReadyToPay(isReadyToPay)` method.

```
  callback.handleIsReadyToPay(true);
```

Payment



A web browser invokes the payment app via an Android intent with payment request information in the intent parameters. The payment app responds with `”methodName”` and `“details”`, which are payment app specific and are opaque to the browser. The browser does not parse the `“details”`. That goes directly to the merchant website.

Payment parameters

- `ArrayList<String> methodNames` - The names of the methods being used. The elements are the keys in `methodData` dictionary. These are the methods that the payment app supports.
- `Bundle[String] methodData` - A mapping from each of the `methodNames` to the output of `JSON.stringify(methodData[methodName].data)`.
- `String merchantName` - The contents of the `<title>` HTML tag of the top-level browsing context on the checkout webpage.
- `String topLevelOrigin` - The scheme-less origin of the top-level browsing context. For example, “https://mystore.com/checkout” is passed as “mystore.com”.
- `Parcelable[] topLevelCertificateChain` - The certificate chain of the top-level browsing context. Null for localhost and file on disk, which are both secure contexts without SSL certificates. Each `Parcelable` is a `Bundle` with a `certificate` key and a byte array value.
- `String paymentRequestOrigin` - The scheme-less origin of the iframe browsing context that invoked `new PaymentRequest(methodData, details, options)` constructor. If the constructor was invoked from top-level context, then the value of this parameter equals the value of `topLevelOrigin` parameter.
- `String total` - The output of `JSON.stringify(details.total.amount)`.
- `String modifiers` - The output of `JSON.stringify(details.modifiers)`, where `details.modifiers` contain only `supportedMethods` and `total`.
- `String paymentRequestId` - The `PaymentRequest.id` field that “push-payment” apps should associate with transaction state. Merchant websites will use this field to query the “push-payment” apps for the state of transaction out of band.

Not all browsers have the capability to determine the values for all of these parameters. Therefore, the payment app should check for existence of these parameters before accessing them.

These parameters are sent to the payment app using intent extras.

```
Bundle extras = new Bundle();
extras.putString("key", "value");
intent.putExtras(extras);
```

If the certificate is not valid in browser judgement, then `PaymentRequest.show()` should not invoke payment apps. Even if the user has chosen to bypass browser’s interstitial warning about that site, PaymentRequest API will be available for manual entry of data by the user, but not for quick and painless payments. Therefore, only a valid certificate chain will be sent to the payment app. This is the entire certificate chain after the browser has resolved it to its root.

(Perhaps surprisingly, the chain that the site serves is not necessarily what the browser ends up validating.)

Payment response

- `int success` - The Activity result of either `RESULT_OK` or `RESULT_CANCELED`, depending on whether the payment app was able to complete its part of the transaction successfully. For example, `success` can be `RESULT_CANCELED` if the user failed to type in the correct PIN code for their account in the payment app.
- `String methodName` - The name of the method being used.
- `String details` - JSON string containing information necessary for the merchant website to complete the transaction. If `success` is `true`, then `details` must be constructed in such a way that `JSON.parse(details)` will succeed.

The response is sent back via Activity.setResult() method.

```
Intent result = new Intent();
Bundle extras = new Bundle();
extras.putString("key", "value");
result.putExtras(extras);
setResult(RESULT_OK, result);  // Change to RESULT_CANCELED on failure.
finish();  // Close the payment activity.
```

If the payment app returns `RESULT_CANCELED`, then the browser may let the user choose a different payment app. The merchant website does not observe this, so there's no need for detailed error codes from the payment app to the merchant website.

Algorithms

This section describes in detail the steps of algorithms that determine the list of possible Android payment apps on the user device. These algorithms fit together as follows:

1. The merchant website provides a list of payment methods in the `PaymentRequest` constructor.
2. The browser finds the locally installed Android payment apps that claim support for the given payment methods.
3. The browser downloads and validates the payment method manifests and the web app manifests for the default applications of the payment method manifests.
4. The browser shows the apps that match the information in these web app manifests.
5. The browser checks for `"*"` in the `"supported_origins"` of the payment method manifests. If found, then the browser shows all matching apps.
6. The browser downloads and validates the default payment method manifests and the default web app manifests for the installed Android payment apps.

7. The browser shows the apps that match the information in these web app manifests.

See authoritative specification in Ingesting payment method manifests algorithm.

Find payment apps

This algorithm queries locally installed Android apps for possible payment apps. It runs when `PaymentRequest.canMakePayment()` or `PaymentRequest.show()` is called.

1. Let `apps` be an empty list of payment apps.
2. If `PaymentMethodData.supportedMethods` contains `"basic-card"` string, then query all apps that can respond to `"org.chromium.intent.action.PAY"` action and have `"basic-card"` in `<meta-data>`. Add these apps to the `apps` list.
3. Let `urlPaymentMethods` be the subset of `PaymentMethodData.supportedMethods` that are valid, absolute URLs with HTTPS scheme.
4. Query all apps that can respond to `"org.chromium.intent.action.PAY"` action with any of the `urlPaymentMethods` in `<meta-data>`. Add these apps to the `apps` list.
5. Remove all apps from the `apps` list that have an empty label.

```
ResolveInfo app;  // Needs to be assigned.
boolean isLabelEmpty = !TextUtils.isEmpty(app.loadLabel(
        getContext().getPackageManager()));
```

6. Return the `urlPaymentMethods` and `apps` lists.

Download payment method manifests

See the authoritative specification in Fetching payment method manifests algorithm.

Validate payment method manifest

See the authoritative specification in Validating and parsing payment method manifests algorithm.

Download web app manifest

See the authoritative specification in Fetching web app manifests algorithm.

Validate web app manifest

The algorithm operates on the contents downloaded in the Download web app manifest algorithm. It returns true for a valid manifest. Here's an example of the contents of the file being parsed:

```
{
   "related_applications": [{
     "platform": "play",
     "id": "com.bobpay.app",
     "min_version": "1",
     "fingerprints": [{
       "type": "sha256_cert",
       "value": "92:5A:39:05:C5:B9:EA:BC:71:48:5F:F2"
     }]
   }]
 }
```

1. Let `manifest` be the output of JSON-parsing the downloaded manifest data, which was decoded as UTF-8.
2. If JSON-parsing fails, then return `false`.
3. If `manifest` is not a dictionary, then return `false`.
4. If `manifest` does not have `"related_applications"` member that is a non-empty list of dictionaries with at least one `"platform": "play"`, then return `false`.
5. For every dictionary in `"related_applications"` with `"platform": "play"`:
    a. If `"id"` is absent, is not a string, or is an empty string, return `false`.
    b. If `"min_version"` is absent, is not a string, or cannot be parsed into a integer, then return `false`.
    c. If `"fingerprints"` is absent, or is not a list, or is an empty list, then return `false`.
    d. For every item in the `"fingerprints"`:
        i. If the `"type"` is not `"sha256_cert"`, then return `false`.
        ii. If the `"value"` is not a string of 32 colon-separated, upper-case hex digits, then return `false`.
6. Return `true`.

Validate payment apps against web app manifests

This algorithm returns `true` if a payment app is allowed to handle payment method, according to a web app manifest.

1. Let `app` be an Android payment app for a `paymentMethodUrl` with a list of downloaded and validated `relatedApplications` (`"related_applications"` with `"platform": "play"` from Validate web app manifest algorithm).
2. For each section in `relatedApplications`, if each of the following conditions is met, then return `true`:
    a. `"id"` equals `app` package name.
    b. `"min_version"` is greater than or equal to `app` version.
    c. The sorted list of the values in `"fingerprints"` equals the sorted list of the SHA256 hash of the certificates of the `app`.
3. Return `false`.

Manifests

See authoritative specification in Manifest format.

The manifests are machine readable files that reside on a server owned by the payment app developer. The locations of these files are derived from the payment method names. For example, if the payment method is called “https://bobpay.com”, then the payment method manifest may be located at https://bobpay.com/payment-method-manifest.json and a corresponding web app manifest may be located at https://bobpay.com/bobpay-app.json. The contents of these files describe the Android apps that are allowed to handle payments for the given payment method.

Example payment method manifest that would be found at https://bobpay.com/payment-method-manifest.json:

```
{"default_applications": ["https://bobpay.com/bobpay-app.json"]}
```

Example web app manifest that would be found at https://bobpay.com/bobpay-app.json:

```
{
  "related_applications": [{
    "platform": "play",
    "id": "com.bobpay.app",
    "min_version": "1",
    "fingerprints": [{
      "type": "sha256_cert",
      "value": "92:5A:39:05:C5:B9:EA:BC:71:48:5F:F2"
    }],
    "url": "https://play.google.com/store/apps/details?id=com.bobpay.app"
  }]
}
```

This fingerprint format is inspired by Digital Asset Links. The file format is an extension of Web App Manifest. This format allows for multiple payment apps, multiple versions of the same app, and multiple operating systems. Android operating system has support for Digital Asset Links since Marshmallow, but web browsers also need to support older versions of Android, so the built-in functionality found in the operating system is not useful.

All of the fingerprints in `"fingerprints"` should match all of the fingerprints in an installed app. To enable multiple versions of the same app with different fingerprints, list each version separately under `"related_applications"`.

The `"min_version"` parameter is the minimum version of the payment app that can be used.

To allow unrestricted use of a payment method identifier, specify `"supported_origins": "*"` in the payment method manifest.

The `"id"`, `"min_version"`, and `"fingerprints"` values are required. The `"id"` value should be non-empty. The `"fingerprints"` list must be non-empty and each dictionary in the list must have `"type"` and `"value"`. The order of the items in `"fingerprints"` is not important. Only `"sha256_cert"` fingerprint type is supported.

The values of `"fingerprints"` can be computed as follows:

```
PackageInfo packageInfo = ...
MessageDigest md = MessageDigest.getInstance("SHA-256")
md.update(packageInfo.signatures[i].toByteArray();
byte[] digest = md.digest();
StringBuilder builder = new StringBuilder(digest.length * 3);
Formatter formatter = new Formatter(builder);
for (byte b : digest) {
    formatter.format(":%02X", b);
}
// Cut off the first ":".
return builder.substring(1);
```

AndroidManifest.xml

Add this in AndroidManifest.xml for the payment app.

```
<manifest package="com.bobpay.app">
  <service android:name=".IsReadyToPayService"
           android:enabled="true"
           android:exported="true">
    <intent-filter>
```

```
      <action android:name="org.chromium.intent.action.IS_READY_TO_PAY" />
    </intent-filter>
  </service>
  <activity android:name=".PaymentActivity"
            android:exported="true">
    <intent-filter>
      <action android:name="org.chromium.intent.action.PAY" />
    </intent-filter>
    <meta-data android:name="org.chromium.default_payment_method_name"
               android:value="https://bobpay.com/put/optional/path/here" />
  </activity>
</manifest>
```

The `"IS_READY_TO_PAY"` service is optional. If there's no such intent handler in the payment app, then the web browser assumes that the app can always make payments.

The activity with the `"PAY"` intent filter should have a `<meta-data>` tag that identifies the default payment method name for the app.

There should be at most one activity that handles `"org.chromium.intent.action.PAY"` and at most one service that handles `"org.chromium.intent.action.IS_READY_TO_PAY"`. These are invoked regardless of the payment method.

To support multiple payment methods, add a `<meta-data>` tag with a `<string-array>` resource.

```
<manifest package="com.bobpay.app">
  <service android:name=".IsReadyToPayService"
           android:enabled="true"
           android:exported="true">
    <intent-filter>
      <action android:name="org.chromium.intent.action.IS_READY_TO_PAY" />
    </intent-filter>
  </service>
  <activity android:name=".PaymentActivity"
            android:exported="true">
    <intent-filter>
      <action android:name="org.chromium.intent.action.PAY" />
    </intent-filter>
    <meta-data android:name="org.chromium.default_payment_method_name"
               android:value="https://bobpay.com/put/optional/path/here" />
    <meta-data android:name="org.chromium.payment_method_names"
               android:resource="@array/my_payment_method_names" />
```

```
  </activity>
</manifest>
```

The resource must be a list of strings. Each string must be a valid, absolute URL with HTTPS scheme. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="my_payment_method_names">
    <item>https://alicepay.com/put/optional/path/here</item>
    <item>https://evepay.com/put/optional/path/here</item>
  </string-array>
</resources>
```

All prefixes are `"org.chromium"`, because W3C is not involved in Chromium's Android-specific APIs.

Note (added September 2017)

Please do not duplicate the default payment method name in the `<string-array>`. If you do, Chrome may become unstable in versions <= 62.

"basic-card"

Any payment app can support "basic-card" payment method. This payment method does not require a payment app manifest. Chrome does not perform signature verification of the payment app that supports only "basic-card". To enable support for this payment method, add the following to AndroidManifest.xml file of the payment app.

```
<manifest package="com.bobpay.app">
  <service android:name=".IsReadyToPayService"
           android:enabled="true"
           android:exported="true">
    <intent-filter>
      <action android:name="org.chromium.intent.action.IS_READY_TO_PAY" />
    </intent-filter>
  </service>
  <activity android:name=".PaymentActivity"
            android:exported="true">
    <intent-filter>
      <action android:name="org.chromium.intent.action.PAY" />
    </intent-filter>
```

```
    <meta-data android:name="org.chromium.default_payment_method_name"
               android:value="basic-card" />
  </activity>
</manifest>
```

Alternatively, “basic-card” can be one of the multiple supported payment methods through the use of a `<resources>` file.

Services

Querying `"IS_READY_TO_PAY"` is one-shot communication without bringing up payment app’s user interface. Messenger fits this paradigm well, but Messenger.sendingUid is available only in newer versions of Android. The alternative call `Binder.getCallingUid()` is not reliable in Messenger. The solution is to use an AIDL.

```
package org.chromium;

interface IsReadyToPayServiceCallback {
    oneway void handleIsReadyToPay(boolean isReadyToPay);
}
```

Save this in `org/chromium/IsReadyToPayServiceCallback.aidl` in your project. The callback is used to enable asynchronous querying.

```
package org.chromium;

import org.chromium.IsReadyToPayServiceCallback;

interface IsReadyToPayService {
    oneway void isReadyToPay(IsReadyToPayServiceCallback callback);
}
```

Save this in `org/chromium/IsReadyToPayService.aidl` in your project. The oneway keyword is necessary to avoid blocking on the call. If querying takes more than 400 ms, the call times out and behaves as if callback.handleIsReadyToPay(**false**); is called. Responding to the `"IS_READY_TO_PAY"` intent works as follows:

```
import org.chromium.IsReadyToPayService;
import org.chromium.IsReadyToPayServiceCallback;

public class IsReadyToPayServiceImpl extends Service {
  private final IsReadyToPayService.Stub mBinder =
    new IsReadyToPayService.Stub() {
```

```
        @Override
        public void isReadyToPay(IsReadyToPayServiceCallback callback) {
            // Check permission here.
            callback.handleIsReadyToPay(true);
        }
    });

  @Override
  public IBinder onBind(Intent intent) {
    return mBinder;
  }
}
```

Permissions

IS_READY_TO_PAY intent

The permission check can be accomplished by checking `Binder.getCallingUid()`. The `onBind()` method in a `Service` is called only once during the lifetime of the `Service`. If multiple apps connect to the `Service` while it's alive, they will all get the same instance. This means, that multiple apps may be talking to same instance of the payment app's `IsReadyToPayService`. Therefore, permission check must happen inside of `isReadyToPay()` call.

```
PackageManager pm = getPackageManager();
Signature[] callerSignatures = pm.getPackageInfo(
        pm.getNameForUid(Binder.getCallingUid()),
        PackageManager.GET_SIGNATURES).signatures;
```

PAY intent

Android intents do not receive a `Message`. Therefore, there's no `sendingUid` to get the name of the package. A payment app should use `Activity.getCallingActivity().getPackageName()` for signature verification in the `"PAY"` intent .

```
Signature[] callerSignatures = getPackageManager().getPackageInfo(
        getCallingActivity().getPackageName(),
        PackageManager.GET_SIGNATURES).signatures;
```

Beware that `getCallingActivity()` is not guaranteed to return an object. Check for null before using its result.