Event Source Notifications in Native Android Applications

fork and edit tutorial (https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/7.1/notifications/push-notifications-overview/push-notifications-native-android-applications/event-source-based-notifications-in-native-android-applications.html) | report issue (https://github.ibm.com/MFPSamples/DevCenter/issues/new)

Overview

Prerequisite: Make sure that you read the Push notifications in native Android applications (../../push-notifications-native-android-applications/) tutorial first.

Event source notifications are notification messages that are targeted to devices with a user subscription. While the user subscription exists, MobileFirst Server can produce push notifications for the subscribed user. These notifications can be delivered by the adapter code to all or some of the devices from which the user subscribed.

To learn more about the architecture and terminology of event-source push notifications refer to the Push notification overview (../../../push-notifications-overview/#notificationTypes) tutorial.

Implementation of the push notification API consists of the following main steps:

On the server side:

- Creating an event source
- Sending notification

On the client side:

- Sending the token and initializing the `WLPush` class
- Registering the event source
- Subscribing to/unsubscribing from the event source

Agenda

- Notification API - server-side
- Notification API - Client-side
- Sample application

Notification API - Server-side

Creating an event source

To create an event source, you declare a notification event source in the adapter JavaScript code at a global level (outside any JavaScript function):

```
WL.Server.createEventSource({
    name: 'PushEventSource',
    onDeviceSubscribe: 'deviceSubscribeFunc',
    onDeviceUnsubscribe: 'deviceUnsubscribeFunc',
    securityTest:'PushApplication-strong-mobile-securityTest
'
});
```

- **`name`** – a name by which the event source is referenced.
- **`onDeviceSubscribe`** – an adapter function that is invoked when the user subscription request is received.
- **`onDeviceUnsubscribe`** – an adapter function that is invoked when the user unsubscription request is received.
- **`securityTest`** – a security test from the `authenticationConfig.xml` file, which is used to protect the event source.

An additional event source option:

```
poll: {
    interval: 3,
    onPoll: 'getNotificationsFromBackend
'
}
```

- **`poll`** – a method that is used for notification retrieval.
  The following parameters are required:
  - **`interval`** – the polling interval in seconds.
  - **`onPoll`** – the polling implementation. An adapter function to be invoked at specified intervals.

Sending a notification

As described previously, notifications can be either polled from the back-end system or pushed by one. In this example, a `submitNotifications()` adapter function is invoked by a back-end system as an external API to send notifications.

```
function submitNotification(userId, notificationText) {
    var userSubscription = WL.Server.getUserNotificationSubscription('PushAdapter.PushEventSource', u
serId);
    if (userSubscription === null) {
        return { result: "No subscription found for user :: " + userId };
    }
    var badgeDigit = 1;
    var notification = WL.Server.createDefaultNotification(notificationText, badgeDigit, {custom:"data"});
        WL.Server.notifyAllDevices(userSubscription, notification);
    return {
        result: "Notification sent to user :: " + userId
    };
}
```

The `submitNotification` function receives the `userId` to send notification to and the `notificationText`.

```
    function submitNotification(userId, notificationText) {
```

A user subscription object contains the information about all of the user's subscriptions. Each user subscription can have several device subscriptions. The object structure is as follows:

```
{
    userId: 'bjones',
    state: {
        customField: 3
    },
    getDeviceSubscriptions: function()[
}
};
```

Next line:

```
var userSubscription = WL.Server.getUserNotificationSubscription('PushAdapter.PushEventSource', use
rId);
```

If the user has no subscriptions for the specified event source, a **null** object is returned.

```
if (userSubscription === null) {
    return {
        result: "No subscription found for user :: " + userI
d
    };
}
```

The `WL.Server.createDefaultNotification` API method creates and returns a default notification JSON block for the supplied values.

```
var badgeDigit = 1;
var notification = WL.Server.createDefaultNotification(notificationText, badgeDigit, {custom:"data"});
```

- **`notificationText`** - The text to be pushed to the device.
- **`Badge`** (number) - A number that is displayed on the application icon or tile (in environments that support it).
- **`custom`** - Custom, or Payload, is a JSON object that is transferred to the application and that can contain custom properties.

The `WL.Server.notifyAllDevices` API method sends notification to all the devices that are subscribed to the user.

```
WL.Server.notifyAllDevices(userSubscription, notification);
```

Several APIs exist for sending notifications:

- `WL.Server.notifyAllDevices(userSubscription, options)` - to send notification to all user's devices.
- `WL.Server.notifyDevice(userSubscription, device, options)` - to send notification to a specific device that belongs to a specific user subscription.
- `WL.Server.notifyDeviceSubscription(deviceSubscription, options)` - to send the notification to a specific device.

Notification API - Client-side

The first step is to create an instance of the `WLClient` class:

```
final WLClient client = WLClient.createInstance(this);
```

You derive all push notification operations from the `WLPush` class.

`getPush` – Use this method to retrieve an instance of the `WLPush` class from the `WLClient` instance.

```
WLPush push = client.getPush();
```

`WLOnReadyToSubscribeListener` – When connecting to MobileFirst Server, the application attempts to register itself with the GCM server to receive push notifications.

```
client.getPush().setOnReadyToSubscribeListener(listener);
client.connect(listener);
```

The `onReadyToSubscribe` method of `WLOnReadyToSubscribeListener` is called when the registration is complete.

```
@Override
public void onReadyToSubscribe() {.....}
```

`WLPush.registerEventSourceCallback`

To register an alias on a particular event source, use the `WLPush.registerEventSourceCallback` method.

The API takes the following arguments:

`alias` - An alias name.
`Adaptername` - Adapter in which the event source is defined.
`EventSourceName` - The event source on which the alias is called.

Example:

```
WLClient.getInstance().getPush().registerEventSourceCallback("myAndroid","PushAdapter","PushEventS
ource",this);
```

Typically, this method is called in the `onReadyToSubscribe` callback function.

```
@Override
public void onReadyToSubscribe() {
   WLClient.getInstance().getPush().registerEventSourceCallback("myAndroid","PushAdapter","PushEven
tSource",this);
}
```

In the Android activity class, override the methods that define the Android activity life cycle as follows:

`onPause()` must call the `setForeground(false)` method of the `WLPush` instance to receive the notification in the notification bar when the application is paused.

```
@Override
protected void onPause() {
   super.onPause();
   if (push != null)
   push.setForeground(false)
;
}
```

`onResume()` must call the `setForeground(true)` method of the `WLPush` instance to receive the notification in the callback of the application.

```
@Override
protected void onResume() {
   super.onResume();
   if (push != null)<
   push.setForeground(true);
}<
```

`onDestroy()` must call the `unregisterReceivers` method of the `WLPush` instance to avoid leak exceptions from the receiver when the application exits.

```
@Override
protected void onDestroy() {
   super.onDestroy();
   if (push != null)
   push.unregisterReceivers()
;
}
```

Subscribing to push notifications

To set up subscription to push notifications, use the `WLPush.subscribe(alias, pushOptions, responseListener)` API.

The API takes the following arguments:

`alias` – The alias to which the device must subscribe.
`pushOptions` – An object of type `WLPushOptions`.
`responseListener` – An object of type `WLResponseListener`, which is called when subscription completes.

Example:

```
WLClient client = WLClient.getInstance();
client.getPush().subscribe("myAndroid",new WLPushOptions(), new MyListener(MyListener.MODE_SUB
SCRIBE));
```

MyListener Implements `WLResponseListener` and provides the following callback functions:

`onSuccess` – Called when subscription succeeds.
`onFailure` – Called when subscription fails.

Unsubscribing from push notifications

To set up unsubscription from push notifications, use the `WLPush.unsubscribe(alias, responseListener)` API.

The API takes the following arguments:

`alias` – The alias to which the device has subscribed.
`responseListener` – An object of type `WLResponseListener`, which is called when unsubscription completes.

Example:

```
WLClient client = WLClient.getInstance();
client.getPush().unsubscribe("myAndroid",new MyListener(MyListener.MODE_UNSUBSCRIBE));
```

MyListener Implements `WLResponseListener` and provides the following callback functions:

- `onSuccess` – Called when unsubscription succeeds.
- `onFailure` – Called when unsubscription fails.

Additional client-side API methods:

`isPushSupported()` - Indicates whether push notifications are supported by the device.

```
WLClient client = WLClient.getInstance();
boolean supported = client.getPush().isPushSupported();
```

`isSubscribed()` - Indicates whether the device is subscribed to push notifications.

```
WLClient client = WLClient.getInstance();
boolean isSubscribed = client.getPush().isSubscribed("myAndroid");
```

Receiving a push notification

When a push notification is received, the `onReceive` method is called on an `WLEventSourceListener` instance.

```
public class MyListener implements WLOnReadyToSubscribeListener, WLResponseListener, WLEven
tSourceListener{
```

The `WLEventSourceListener` instance is registered during the `registerEventSourceCallback` callback.

```
WLClient.getInstance().getPush().registerEventSourceCallback("myAndroid", "PushAdapter","PushEvent
Source", this );
```

The `onReceive` method displays the received notification on the screen.

```
@Override
public void onReceive(String arg0, String arg1) {
   AndroidNativePush.updateTextView("Notification received  " + arg0)
;
}<
```

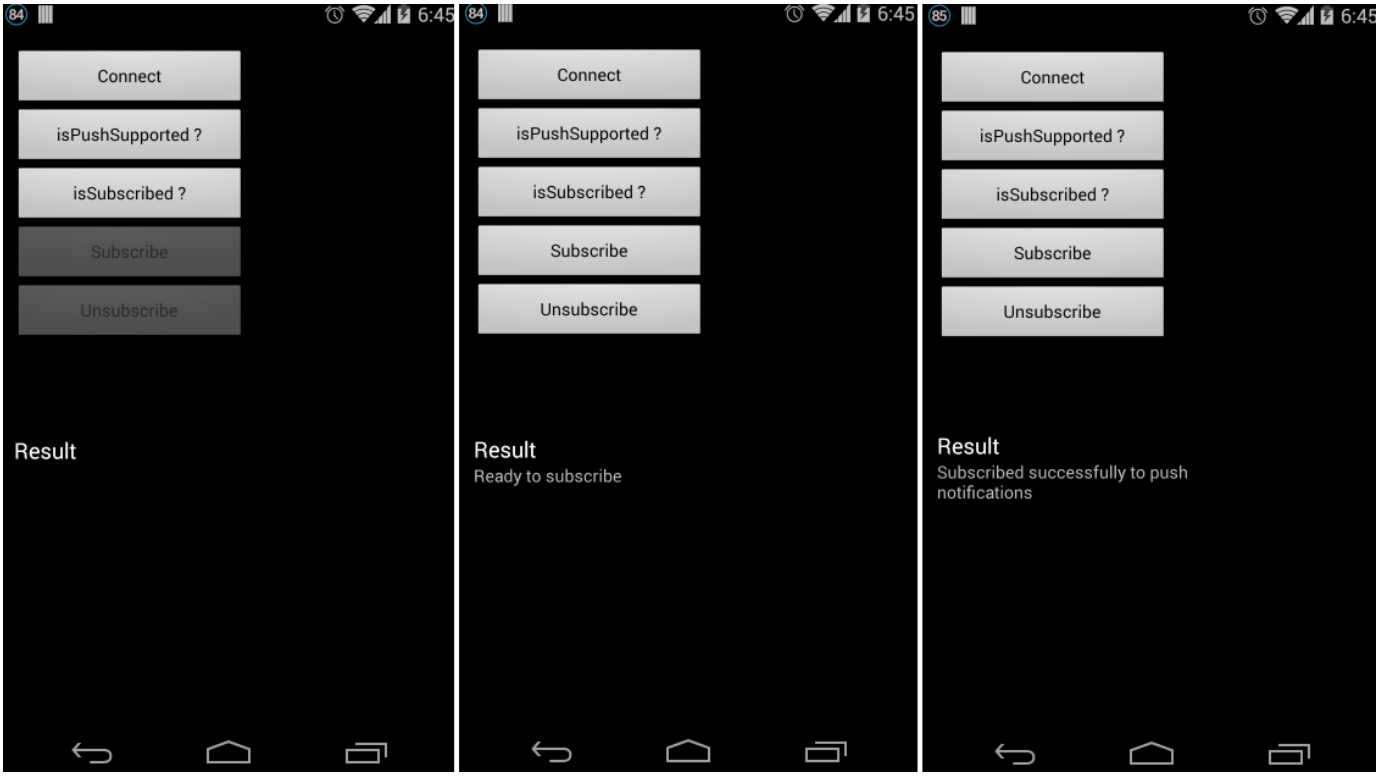
If the application is not running, the notification icon appears on the notification bar at the top of the screen.

Sample application

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/EventSourceNotifications/tree/release71) the MobileFirst project.

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/EventSourceNotificationsAndroid/tree/release71) the Native project.

- The `EventSourceNotifications` project contains a MobileFirst native API that you can deploy to your MobileFirst Server instance.
- The `EventSourceNotificationsAndroid` project contains a native android application that uses a MobileFirst native API library to subscribe to push notifications and receive notifications from GCM.
- Make sure to update the `wlclient.properties` file in the native project with the relevant server settings.



Sending a notification

To test the application is able to receive a push notification you can perform one of the following:

1. Right-click the adapter in MobileFirst Studio and select **Call MobileFirst Adapter**
2. If using the CLI, for example:

```
$ mfp adapter call
[?] Which endpoint do you want to use? PushAdapter/submitNotification
[?] Enter the comma-separated parameters: "the-user-name", "hello!"
[?] How should the procedure be called? GET
```