WOPaypal Framework

Overview

Travis Cripps

I guess the first question we should cover is if you're using PayPal's Instant Payment Notification service to receive the notification from PayPal that your user has completed some type of PayPal transaction. If you're not using that, but will instead log into PayPal's site every day and process the transactions, then you don't need to worry about the Listener and the delegates, nor indeed the whole process of getting the request parameters from PayPal. You could then just use the button or link components to either start a transaction for a single item or add the item to a PayPal shopping cart. You could also pretty much skip the rest of this email. 😉

However, if you are interested in processing the transactions through PayPal's IPN (instant payment notification) mechanism, you can look at the ipnAction method in PayPalAction.java source code in the WOPayPal framework to see exactly how I receive their request (notification) and validate it by echo-ing their transaction key back to PayPal. After receiving the verification, PayPal will send the real message. That's the content you need to handle. There are several return codes they'll give, and you'll need to be able to handle them properly, but it sounds as though you're prepared to do so.

If I might offer my opinion, you may find it advantageous to use the existing code rather than rewrite your own request handling. You could do this by copying my ipnAction and adapting it to your needs, or by creating your own delegate class that implements a few methods and assigning your delegate to handle the various response codes returned by PayPal. Don't be thrown off by the assignment of the delegate as you mentioned above. Specifically, the PayPalNotificationLogger is a just a simple class that I created as a sample delegate. All it does is "log" the messages it receives to the System.out using NSLog. In your application, you'd create your own delegate class implementation that would do something more useful.

Let me briefly try to convince you of the benefits of notifications and delegates...

A Success Story

Imagine a small one-man company named ACME. ACME takes orders from its customers and sends them along to its supplier, ZYX. The supplier then phones ACME to confirm that the order was received and give some details about when ACME can expect its shipment. As the company grows, the CEO gets too many responsibilities to continue to interface with the ZYX and handle all the order management, so he hires an assistant and installs a PBX switchboard to route calls from ZYX to his assistant. The CEO teaches his new assistant the responsibilities of order management and assigns the assistant to "handle" those responsibilities. As a result of its new efficiency, ACME grows, all the employees gets rich from their stock options, and everyone lives happily ever after.

Decoding the Story

The above (fairly contrived) example demonstrates the general idea of notifications and delegates. You could say that the callback from supplier ZYX is analogous to the PayPal IPN. The switchboard in this case is the equivalent of an NSNotificationCenter; it detects an incoming call and then directs the call (the NSNotification message) to the appropriate listener. In this case, the switchboard is programmed to detect a call from company ZYX and forward it to the assistant who has been assigned, or "delegated," the responsibility of handling calls from ZYX. Once the assistant, (the delegate) receives the call, it does one of the things it knows how to do with the call, depending on the type of message it receives.

Basic Definitions

Back to reality... Notifications and delegates are simple design patterns that you'll see scattered throughout Cocoa and WebObjects. They're very useful, and not really too complicated.

Basically, a notification is a way to pass messages around without having to know the details of the class that will receive them. Notifications are "broadcasted" by the NSNotificationCenter, a controller object which handles sending the notification messages to the appropriate recipient(s). You "subscribe" to the notification center by asking it to send notifications of a certain type to the subscriber object. Typically, you ask for a notification by name. You can also send a message to the notification center to tell it to broadcast a notification message to all interested parties.

Similarly, a delegate is a way to pass the responsibility for processing certain off to another class. Using delegates allows you to "loosely couple" classes together and customize behavior by assigning the delegate class you need in a particular instance. The delegate need only implement a certain set of methods so that it can handle the messages you're going to pass to it. You could even have multiple delegates with different behavior and swap them as necessary at runtime.

How It Relates to WOPayPal

I use a notification mechanism to broadcast the message to handle different conditions of the PayPal notification to the "listener" (subscribed to receive certain notifications) which then forwards the message to the assigned "delegate." In the case of WOPayPal, the delegate just has to implement the methods to handle the PayPal messages that you're interested in. There are 5 methods in the delegate interface:

- public void processDeniedPaypalTransaction(WORequest aRequest);
- public void processFailedPaypalTransaction(WORequest aRequest);
- public void processInvalidPaypalTransaction(WORequest aRequest);
- public void processPendingPaypalTransaction(WORequest aRequest);
- public void processValidPaypalTransaction(WORequest aRequest);

So, basically, all you need to do to get the framework to work for you is:

- Define a class that contains any of the above methods. This class is your delegate class to process the messages about PayPal notifications.

- Tell the listener (the object subscribed to the notifications) to use your delegate rather than the simple default delegate that only logs the request to the System.out. I do this in my Application class' constructor method. Just call the static method on PayPalNotificationListener, passing an instance of your delegate class as the argument.

```
PayPalNotificationListener.setDelegate(new myPayPalDelegate());
```

- If you want to return a custom component after a successful or a cancelled transaction, set a property for each in the launch arguments or the Properties file of your application.

```
SuccessfulPayPalTransactionComponent = MySuccessPage
CancelledPayPalTransactionComponent = MyCancelledTransactionPage
```

These pages will be returned by the DirectAction methods that PayPal will call in the case of a successful transaction or a cancelled transaction. To use these values, set the returnURL and cancelURL values in the various hyperlink and button components to the url of the PayPalAction DirectActions in your application, e.g. http://yoursite.com/cgi-bin/WebObjects/YourApp.woa/wa/PayPalAction/return or http://yoursite.com/cgi-bin/WebObjects/YourApp.woa/wa/PayPalAction/cancel.

- Tell PayPal how to find your application and where to send the IPN messages in your IPN preferences.

```
http://yoursite.com/cgi-bin/WebObjects/YourApp.woa/wa/PayPalAction/ipn
```

Don't forget that the IPN feature have to connect to a WO app reachable from a public IP!

Sandbox mode

Starting in Wonder builds after mid-September 2009, you can use WOPayPal with the PayPal Sandbox. If you set this property :

```
er.wopaypal.sandboxmode=true
```

All urls will use www.sandbox.paypal.com instead of www.paypal.com. Don't forget to register for a sandbox account!

Conclusion

That's pretty much the story of WOPayPal. I hope that this helps clarify what's going on in the framework, and how you can easily leverage it. At the very least, if you have a copy of the source code, you can copy out the relevant portions of the DirectAction class. Please let me know if you have further questions. I promise not to be so long-winded in the future.