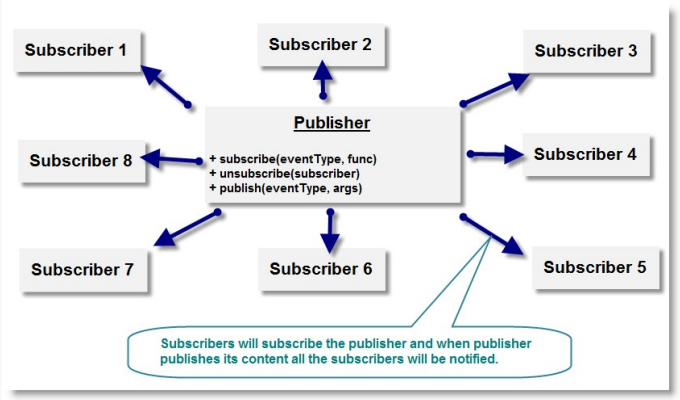
JavaScript Design Pattern: Publisher Subscriber Pattern

Published 4/3/2013 by Raghav Khunger

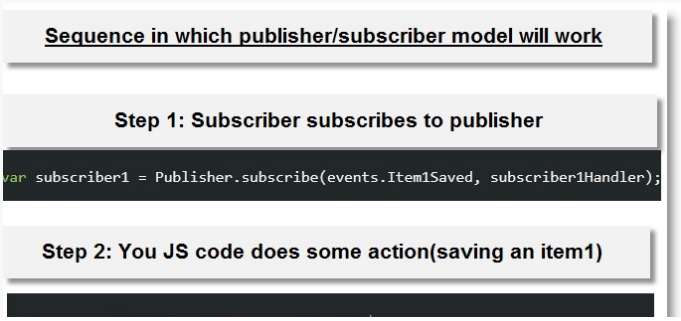
Introduction

This pattern enables to build agile software architecture by decoupling items/objects which produce information and items/objects which consume it. The publisher/subscriber pattern is a software design pattern in which an object, called the subject, maintains a list of its observers, and notifies them automatically of any state changes. It is mainly used to broadcast information to multiple subscribers/observers. The essence of this pattern is to allow one object (the observer, publisher) to watch another (the subject, the subscriber). Through this pattern subscribers can register to receive events from the publisher. When the publisher requires informing its subscribers of its state change, it simply sends the event to each subscriber. The publisher doesn't need to know anything special about its subscribers. Below I will discuss how to implement this pattern in real world usage. Thanks to AddyOsmani for his article from which I got the idea for writing this article.



When do we need this pattern?

- When a change to one object needs changing other objects.
- When an object should be able to notify other objects without making presumption about the other objects (loose coupling).



Let's create a JavaScript file named "MyNameSpace.js" this file will contain the global namespace used by the JS code.

MyNamespace.js

```
if (typeof (MyNamespace) === 'undefined' || MyNamespace === null) {
    MyNamespace = {};
}
```

Create a new JS file named "MyNamespace.Enums.js". This file will be used to define eventtypes. Instead of writing hardcoded event types I decided to make an enum for it.

MyNamespace.Enums.js

```
/// <reference path="MyNamespace.js" />

if (typeof (MyNamespace) === 'undefined' || MyNamespace === null) {
    MyNamespace = {};
}
if (typeof (MyNamespace.Enums) === 'undefined' || MyNamespace.Enums === null) {
    MyNamespace.Enums = {};
}

MyNamespace.Enums.MyClassEvents = function () {
    return {
        Item1Saved: 0,
        Item2Saved: 1,
        Item3Saved: 2
    };
} ();
```

The next file *Publisher.js* is the heart of the pattern discussed in this article, the *Publisher,* which is responsible to subscribe, unsubcribe, publishes/notify the subscribers.

Publisher.js:

```
Ref: http://msdn.microsoft.com/en-us/scriptjunkie/hh201955.aspx by AddyOsmani
if (typeof (Publisher) === 'undefined' || Publisher === null) {
```

```
    Publisher = {};
}
Publisher = (function () {
    'use strict';
    var eventTypes = {},
        subscriberID = -1;

    var subscribe = function (eventType, func) {
        if (!eventTypes[eventType]) {
            eventTypes[eventType] = [];
        }
        var subscriber = (++subscriberID).toString();
        eventTypes[eventType].push({
            subscriber: subscriber,
            func: func
        });
        return subscriber;
    };

    var unsubscribe = function (subscriber) {
        var i = eventTypes.length;
        while (i--) {
            if (eventTypes[i]) {
                var j = eventTypes[i].length;
                while (j--) {
                    if (eventTypes[i][j].subscriber === subscriber) {
                        eventTypes[i].splice(j, 1);
                        return subscriber;
                    }
                }
            }
        }
        return false;
    };

    var publish = function (eventType, args) {

        if (!eventTypes[eventType]) {
            return false;
        }

        setTimeout(function () {
            var subscribers = eventTypes[eventType],
                len = subscribers ? subscribers.length : 0;

            while (len--) {
                subscribers[len].func(eventType, args);
            }
        }, 0);
        return true;
    };

    return {
        publish: publish,
        subscribe: subscribe,
        unsubscribe: unsubscribe
```

```
    };
} ());
```

Implementation

Now let's come the implementation on this topic. We have a custom JS class which is used to save some items in database. Now when these items are saved will publish this information so that the other subscribers get to know this information.

MyNamespace.Utilities.js

```
/// <reference path="MyNamespace.js" />
/// <reference path="EventsEnum.js" />
/// <reference path="Observer.js" />

if (typeof (MyNamespace.Utilities) === 'undefined' || MyNamespace.Utilities === null) {
    MyNamespace.Utilities = {};
}
MyNamespace.Utilities = (function () {
    'use strict';
    var myClass = (function () {
        var my = {};
        var privateVariable1;
        var privateFunction1 = function () {

        };
        var events = MyNamespace.Enums.MyClassEvents;
        my.saveItem1 = function () {
            //do operation...
            console.log('saveItem1 fired');
            Publisher.publish(events.Item1Saved, 'Item1');
        };
        my.saveItem2 = function () {
            //do operation...
            console.log('saveItem2 fired');
            Publisher.publish(events.Item2Saved, 'Item2');
        };
        my.saveItem3 = function () {
            //do operation...
            console.log('saveItem3 fired');
            Publisher.publish(events.Item3Saved, 'Item3');
        };
        return my;
    } ());

    return {
        MyClass: myClass
    };
} ());
```

Now come to UI side, the html page.

Sample.html

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Publisher/Subscriber Sample</title>
    <script src="Scripts/MyNamespace.Enums.js" type="text/javascript"></script>
    <script src="Scripts/Publisher.js" type="text/javascript"></script>
    <script src="Scripts/MyNamespace.Utilities.js" type="text/javascript"></script>
    <script type="text/javascript">

        var subscriber1Handler = function (eventType, data) {
            console.log('subscriber1' + ', ' + eventType + ', ' + data);
        };
        var subscriber2Handler = function (eventType, data) {
            console.log('subscriber2' + ', ' + eventType + ', ' + data);
        };
        var subscriber3Handler = function (eventType, data) {
            console.log('subscriber3' + ', ' + eventType + ', ' + data);
        };

        var events = MyNamespace.Enums.MyClassEvents;

        var subscriber1 = Publisher.subscribe(events.Item1Saved, subscriber1Handler);
        var subscriber2 = Publisher.subscribe(events.Item2Saved, subscriber2Handler);
        var subscriber3 = Publisher.subscribe(events.Item3Saved, subscriber3Handler);


        MyNamespace.Utilities.MyClass.saveItem1();
        MyNamespace.Utilities.MyClass.saveItem2();
        MyNamespace.Utilities.MyClass.saveItem3();

    </script>
</head>
<body>
</body>
</html>
```

How *Subscribers* will subscribe to *Publisher*?

You need to write "Publisher.subscribe(*<event type>*,*<callback handler>*)" in order to make the *Subscriber* subscribe to *Publisher* for that particular *eventtype.* When *Publisher* publishes some content for that particular *eventtype* the *Subscriber* will be notified.

```
var subscriber1 = Publisher.subscribe(events.Item1Saved, subscriber1Handler);
var subscriber2 = Publisher.subscribe(events.Item2Saved, subscriber2Handler);
var subscriber3 = Publisher.subscribe(events.Item3Saved, subscriber3Handler);
```

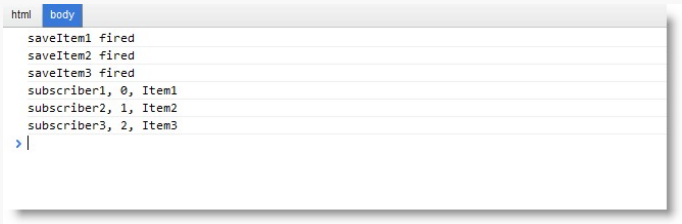
How *Publisher* will publish?

You need to write "Publisher.publish(*<eventtype>*, *<args>*);" in order to make the *Publisher* publishes its content. In the below image *Publisher* has published the information about the *item1*

saved. All the subscribers which are subscribed to *Publisher* for *saveItem1* eventtype will be notified and they can do the actions according to their event handlers.

```
    };
    var events = MyNamespace.Enums.MyClassEvents;
    my.saveItem1 = function () {
        //do operation...
        console.log('saveItem1 fired');
        Publisher.publish(events.Item1Saved, 'Item1');
    };
    my.saveItem2 = function () {
        //do operation...
        console.log('saveItem2 fired');
        Publisher.publish(events.Item2Saved, 'Item2');
    };
    my.saveItem3 = function () {
        //do operation...
        console.log('saveItem3 fired');
        Publisher.publish(events.Item3Saved, 'Item3');
    };
    return my;
} ());
```

Output after running the above code



Conclusion

Based on the discussion of above pattern it should be evident that this pattern provides an ideal mechanism to notify other objects about the state of one object with tight coupling the two objects. It helps us to make a platform for broadcast communification. The notification is broadcast to all the intersted objects who want to listen to that broadcast.

Download Sample: PubSubDemo

tags : javascript design pattern: publisher subscriber pattern, javascript publisher subscriber, javascript publisher subscriber pattern