JS Patterns - Publication & Subscription

- Dr Nick Hayward

A brief overview of implementing the *PubSub* pattern using plain JavaScript.

contents

intro

A variation of the standard *observer* pattern is the *publication and subscription*, or *PubSub*, pattern.

Commonly used in JavaScript development, the *PubSub* pattern publishes a *topic* or *event* channel. This publication acts as a mediator between *subscriber* objects wishing to receive notifications, and the *publisher* object announcing an event.

This mediator, or event system, allows us as developers to easily define specific events, which may then pass custom arguments to a subscriber.

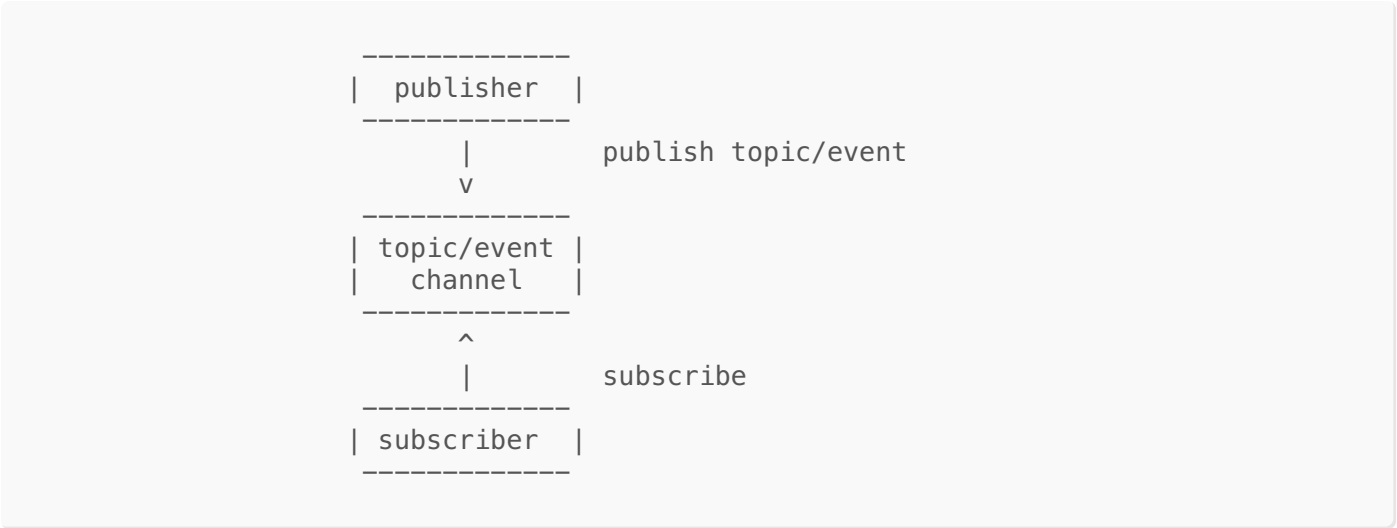
In effect, we're trying to avoid potential dependencies between *subscriber* objects and the *publisher* object.

Inherent to this pattern is the simple abstraction of responsibility. Publishers are unaware of the nature or type of subscribers for their messages. Likewise, subscribers are unaware of the specifics for a given publisher. Subscribers simply identify their interest in a given topic or event, and then receive notifications of updates for a given subscribed channel.

A primary difference with the *observer* pattern is the abstraction of the *subscriber*. As long as a *subscriber* is able to handle and receive the notifications, they can use the broadcasts by the *publisher*.

So, we're creating an *event* system that sits between the publisher and the subscriber.

e.g.



In effect, the subscriber is now registered to listen for certain *topic* or *event* announcements. The handler for a given subscriber may then decide to output the return from the publisher, or perhaps simply log the update, and so on.

benefits of observer & pub-sub patterns

Observer and PubSub patterns help us, as developers, consider more carefully the inherent relationships that exist within our app's logic and structure.

We should be able to identify certain aspects of our application, which contain direct relationships. Many of these dependencies may be replaced with subjects and observers.

A change to tightly coupled code may initially seem straightforward, but in a large application this can quickly become bad practice. A seemingly minor change to a tightly coupled application will often create a cascade or waterfall effect for subsequent required changes and updates.

A known side-effect of such tightly-coupled code is the frequent need to *mock* usage &c. in tests. Again, as the app scales this will be increasingly fraught with issues, including testing complexity and time requirements.

With the PubSub pattern, an inherent benefit for our code is the creation of smaller, loosely coupled blocks, which may then help improve management and reuse.

example

event system

```
// constructor for pubsub object
function PubSub () {
this.pubsub = {};
}

// publish - expects topic/event & data to send
PubSub.prototype.publish = function (topic, data) {
  // check topic exists
  if (!this.pubsub[topic]){
    console.log(`publish - no topic...`);
    return false;
  }
  // loop through pubsub for specified topic - call subscriber functions...
  this.pubsub[topic].forEach(function(subscriber) {
      subscriber(data || {});
    });
};

// subscribe - expects topic/event & function to call for publish notification
PubSub.prototype.subscribe = function (topic, fn) {
  // check topic exists
  if (!this.pubsub[topic]) {
    // create topic
    this.pubsub[topic] = [];
    console.log(`pubsub topic initialised...`);
  }
  else {
    // log output for existing topic match
    console.log(`topic already initialised...`);
  }
  // push subscriber function to specified topic
  this.pubsub[topic].push(fn);
};
```

usage

```
// basic log output
var logger = data => { console.log( `logged: ${data}` ); };

// test function for subscriber
var domUpdater = function (data) {
  document.getElementById('output').innerHTML = data;
}

// instantiate object for PubSub
const pubSub = new PubSub();

// subscriber tests
pubSub.subscribe( 'test_topic', logger );
pubSub.subscribe( 'test_topic2', domUpdater );
pubSub.subscribe( 'test_topic', logger );

// publisher tests
pubSub.publish('test_topic', 'hello subscribers of test topic...');
pubSub.publish('test_topic2', 'update notification for test topic2...');
```

- Demo - Pub/Sub