jQuery Design Patterns

jQuery is a feature-rich JavaScript library that makes HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a variety of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

The book starts off with a refresher to jQuery and will then take you through the different design patterns, such as facade, observer, publisher/subscriber, and so on. We will also learn about client-side templating techniques and libraries, as well as some plugin development patterns. Finally, we will look into some best practices and optimization tips that you can use to make the best of jQuery.

Who this book is written for
This book is for existing jQuery developers or new developers who want to get an understanding of how the various industry standard patterns can be applied to jQuery applications, help large teams collaborate and create well organized and extendable implementations.

What you will learn from this book
- Observe for simple and delegated user events
- Achieve greater flexibility and code decoupling
- Have a central point for emitting and receiving application level events
- Structure the application into small independent modules
- Abstract complex APIs
- Isolate the procedure of generating complex parts of the application
- Efficiently orchestrate asynchronous procedures using jQuery Deferred and Promises
- Utilize the most widely-used client-side templating libraries for more complex use cases

jQuery Design Patterns

Learn the best practices on writing efficient jQuery applications to maximize performance in large-scale deployments

Thodoris Greasidis

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'The Publish/Subscribe Pattern'
- A synopsis of the book’s content
- More information on jQuery Design Patterns
Thodoris Greasidis is a senior web engineer from Greece. He graduated with honors from the University of Thessaly, holds a polytechnic diploma in computer, networking, and communications engineering, and a master's degree in computer science. He is a full-stack developer, responsible for implementing large-scale web applications with intuitive interfaces and high-availability web services.

Thodoris is part of the Angular-UI team and has made many open source contributions, with a special interest in Mozilla projects. He is also an active member of the Athens AngularJS Meetup and a technical reviewer of Mastering jQuery UI, Packt Publishing.

He is a JavaScript enthusiast and loves bitwise operations. His interests also include NodeJS, Python, project scaffolding, automation, and artificial intelligence, especially multi-agent systems.
Preface

Since its introduction in 2006, the jQuery library has made DOM traversals and manipulations much easier. This has resulted in the appearance of Web pages with increasingly complex user interactions, thus contributing to the maturing of Web as a platform capable of supporting large application implementations.

This book presents a series of best practices that make the implementation of Web applications more efficient. Moreover, we will analyze the most important Design Patterns that Computer Science has to offer, which can be applied to Web development. In this way, we will learn how to utilize techniques that are thoroughly used and tested in other fields of programming, which were initially created as generic methods to model solutions of complex problems.

In jQuery Design Patterns, we will analyze how various Design Patterns are utilized in the implementation of jQuery and how they can be used to improve the organization of our implementations. By adopting the Design Patterns demonstrated in this book, you will be able to create better organized implementations that resolve large problem categories faster. Moreover, when used by a developer team, they can improve the communication between them and lead to homogenous implementation, where every part of the code is easily understood by others.

What this book covers

Chapter 1, A Refresher on jQuery and the Composite Pattern, will teach the reader how to write the code using the Composite Pattern and method chaining (Fluent Interface) by analyzing how they are used for the implementation of jQuery itself. It also demonstrates the Iterator Pattern that nicely pairs with the Composite Collection objects that jQuery returns.
Chapter 2, The Observer Pattern, will teach you how to respond to user actions using the Observer Pattern. It also demonstrates how to use Event Delegation as a way to reduce the memory consumption and complexity of the code that handles dynamically injected page elements. Finally, it will teach you how to emit and listen for Custom Events in order to achieve greater flexibility and code decoupling.

Chapter 3, The Publish/Subscribe Pattern, will teach you how to utilize the Pub/Sub Pattern to create a central point to emit and receive application-level events, as a way to decouple your code and business logic from the HTML that is used for presentation.

Chapter 4, Divide and Conquer with the Module Pattern, demonstrates and compares some of the most commonly used Module Patterns in the industry. It will teach you how to structure your application in small independent Modules using Namespacing, leading to expandable implementations that follow the Separation of Concerns principle.

Chapter 5, The Facade Pattern, will teach you how to use the Facade Pattern to wrap complex APIs into simpler ones that are a better match for the needs of your application. It also demonstrates how to change parts of your application, while keeping the same module-level APIs and avoid affecting the rest of your implementation.

Chapter 6, The Builder and Factory Patterns, explains the concepts of and the differences between the Builder and Factory Patterns. It will teach you how and when to use each of them, in order to improve the clarity of your code by abstracting the generation of complex results into separate dedicated methods.

Chapter 7, Asynchronous Control Flow Patterns, will explain how jQuery's Deferred and Promise APIs work and compare them with the classical Callbacks Pattern. You will learn how to use Promises to control the execution of asynchronous procedures to run either in an order or parallel to each other.

Chapter 8, Mock Object Pattern, teaches you how to create and use Mock Objects and Services as a way to ease the development of your application and get a sense of its functionality, long before all its parts are completed.

Chapter 9, Client-side Templating, demonstrates how to use the Underscore.js and Handlebars.js templating libraries as a better and faster way to create complex HTML structures with JavaScript. Through this chapter, you will get an overview of their conventions, evaluate their features, and find the one that best matches your taste.
Chapter 10, Plugin and Widget Development Patterns, introduces the basic concepts and conventions of jQuery Plugin development and analyzes the most commonly used design patterns, so that you will be able to identify and use the best match for any use case.

Chapter 11, Optimization Patterns, guides you with the best tips to create a highly efficient and robust implementation. You will be able to use this chapter as a checklist of best practices that improve the performance and lower the memory consumption of your applications, before moving them to a production environment.
In this chapter, we will showcase the Publish/Subscribe Pattern, a design pattern quite similar to the Observer Pattern but with a more distinct role that is a better fit for more complex use cases. We will see how it differs from the Observer Pattern and how jQuery adopted some of its concepts and brought them to its Observer Pattern implementation.

Later, we will proceed and rewrite our previous chapter's example using this pattern. We will use this pattern's benefits to add some extra features and also reduce the coupling of our code with the elements of the web page.

In this chapter, we will:

- Introduce the Publish/Subscribe Pattern
- Learn how it differs and what advantages it has over the Observer Pattern
- Learn how jQuery brings some of its features to its methods
- Learn how to emit custom events with jQuery
- Rewrite and extend the example from Chapter 2, The Observer Pattern, using this pattern
Introducing the Publish/Subscribe Pattern

The Publish/Subscribe Pattern is a Messaging Pattern where the emitters of the messages, called the **publishers**, multicast messages to a number of recipients, called the **subscribers**, that have expressed their interest in receiving such messages. The key concept of this pattern, which is also commonly referred to as the Pub/Sub Pattern in short, is to provide a way to avoid dependencies between the publishers and their subscribers.

An extra concept of this pattern is the use of **topics** that are used by the subscribers in order to express that they are only interested in messages of a specific type. This way, publishers filter subscribers before sending a message and distribute that message only to the appropriate ones, thereby reducing the amount of traffic and work required on both sides.

Another common variant is to use a central, application-wide object, known as the **broker**, that relays messages produced by the publishers to the relevant subscribers. The broker, in this case, acts as a well-known message handler to send and subscribe to message topics. This enables us, instead of coupling different application parts together, to only reference the broker itself and also the topic that our components are interested in. Even though topics might not be an absolute requirement in the first variant of this pattern, this variant plays an essential role in scalability since there will commonly exist way less brokers (if not just one) than publishers and subscribers.
By following a subscription scheme, the code of the publisher is completely decoupled from the subscribers, meaning that the publisher does not have to know the objects depend on them. As a result, we do not need to hard code to the publisher each separate action that should be executed on the different parts of our application. Instead, the components of an application, and possibly third-party extensions, subscribe to be notified only about topics/events that they need to know. In such distributed architecture, adding a new feature to an existing application requires minimal to no changes to the application components it depends on.

How it differs from the Observer Pattern
The most basic difference is that, by definition, the Pub/Sub Pattern is a one-way-Messaging Pattern that can also pass a message, unlike the Observer Pattern that just describes how to notify the observers about a specific state change on the subject.

Moreover, unlike the Observer Pattern, the Pub/Sub Pattern with a broker results in more loosely coupled code for the different parts of an implementation. This is because the observers need to know their subject that is emitting the events; however, on the other hand, the publishers and their subscribers only need to know the broker that is used.
How it is adopted by jQuery

Once again, the jQuery library provides us with a convenient way to take advantage of the Pub/Sub Pattern in our code. Instead of extending its API by adding new methods specifically named "publish" and "subscribe" and introducing new concepts, the developers decided to extend the jQuery.fn.on() and jQuery.fn.trigger() methods with the ability to handle and emit custom events. This way, jQuery can be used to implement a publisher/subscriber communication scheme using the already known convenient methods it provides.

Custom events in jQuery

Custom events allow us to use almost any user-defined string value as a common event that we can add listeners for, and also manually fire it on page elements. As an extra but a precious feature, custom events can also carry some extra data to be delivered to the listeners of the event.

The jQuery library added its own custom events implementation, before it was actually added to any web specification. This way, it was proved how useful they can be when used in web development. As we saw in the previous chapter, in jQuery, there is a specific part of the implementation that handles both the common element event and also custom events. The jQuery.event object holds all the internal implementations related to firing and listening to events. Also, the jQuery.Event class is a dedicated wrapper that jQuery uses for the needs of both the common element events and its custom events implementation.

Implementing a Pub/Sub scheme using custom events

In the previous chapter, we saw how the jQuery.fn.on() method can be used to add event listeners on elements. We also saw that its implementation is maintaining lists with the added handlers and notifying them when required. Moreover, the event name seems to have the same coordination purpose, just like the topic. This implementation semantics seem to match exactly with the Pub/Sub Pattern as well.
The jQuery.fn.trigger() method actually uses the internal jQuery.event.
trigger() method that is used to fire events in jQuery. It iterates over the internal
handlers list and executes them with the requested event along with any extra
parameters that the custom event defines. Once again, this also matches the
operation requirements of the Pub/Sub Pattern.

As a result, jQuery.fn.trigger() and jQuery.fn.on() seem to match the
needs of the Pub/Sub Pattern and can be used instead of separate "publish" and
"subscribe" methods, respectively. Since they are both available on the jQuery.fn
object, we can use these methods on any jQuery object. This jQuery object will act
as an intermediate entity between the publishers and the subscribers, in a way that
perfectly aligns with the definition of the broker.

A good common practice, which is also used by a lot of jQuery plugins, is to use
the outermost page element that holds the implementation of the application or
the plugin as the broker. On the other hand, jQuery actually allows us to use any
object as a broker, since all that it actually needs is a target to emit an observe for our
custom events. As a result, we could even use an empty object as our broker such
as $({}), in case using a page element seems too restricting or not clean enough
according to the Pub/Sub Pattern. This is actually what the jQuery Tiny Pub/Sub
library does, along with some method aliasing, so that we actually use methods
named "publish" and "subscribe" instead of jQuery's "on" and "trigger". For more
information on Tiny, you can visit its repository page at https://github.com/
cowboy/jquery-tiny-pubsub.

## Demonstrating a sample use case

In order to see how the Pub/Sub Pattern is used, and make it easy to compare it with
the Observer Pattern, we are going to rewrite the dashboard example from Chapter 2,
The Observer Pattern, using this pattern. This will also clearly demonstrate how this
pattern can help us decouple the individual parts of an implementation and make it
more extendable and scalable.
Using Pub/Sub on the dashboard example

For the needs of this demonstration, we will use the HTML and CSS files exactly as we saw them in Chapter 2, The Observer Pattern.

Dashboard Example

To apply this pattern, we will only need to change the code in the JavaScript file with our new implementation. In the following code snippet, we can see how the code was changed in order to adapt to the Publisher/Subscriber Pattern:

```javascript
$(document).ready(function() {
    window.broker = $('.dashboardContainer');

    $('#categoriesSelector').change(function() {
        var $selector = $(this);
        var message = { categoryID: $selector.val() };
        broker.trigger('dashboardCategorySelect', [message]);
    });

    broker.on('dashboardCategorySelect', function(event, message) {
        var $dashboardCategories = $('.dashboardCategory');
        var selectedIndex = +message.categoryID;
        var $selectedItem = $dashboardCategories.eq(selectedIndex).show();
        $dashboardCategories.not($selectedItem).hide();
    });

    $('.dashboardCategory').on('click', 'button', function() {
        var $button = $(this);
        var message = { categoryName: $button.text() };
        broker.trigger('categoryItemOpen', [message]);
    });
});
```
Just like in our previous implementation, we use $(document).ready() in order to delay the execution of our code until the page has been fully loaded. First of all, we declare our broker and assign it to a new variable on the window object so that it is globally available on the page. For our application's broker, we are using a jQuery object with the outermost container of our implementation, which in our case is the <div> element with the dashboardContainer class.

Even though using global variables is generally an anti-pattern, we store the broker into a global variable since it is an important synchronization point of the whole application and must be available for every piece of our implementation, even to those that are stored in separate .js files. As we will discuss in the next chapter about the Module Pattern, the preceding code could be improved by storing the broker as a property of the application's namespace.
In order to implement the category selector, we are first observing the `<select>` element for the `change` event. When the selected category changes, we create our message using a plain JavaScript object with the `value` of the selected `<option>` stored in the `categoryID` property. Then, we publish it in the `dashboardCategorySelect` topic using the jQuery `jQuery.fn.trigger()` method on our broker. This way, we move from a UI element event to a message with application semantics that contains all the required information. Right below, in our subscriber's code, we are using the `jQuery.fn.on()` method on our broker with the `dashboardCategorySelect` topic as a parameter (our custom event), just like we would do to listen for a simple DOM event. The subscriber then uses the `categoryID` from the received message, just like we did in the implementation of the previous chapter, to display the appropriate category items.

Following the same approach, we split the code that handles adding and closing information boxes in our dashboard in publishers and subscribers. For the needs of this demonstration, the message of the `categoryItemOpen` topic contains just the name of the category we want to open. However, in an application where the box content is retrieved from a server, we would probably use a category item ID instead. The subscriber then uses the category item name from the message to create and insert the requested information box.

Similarly, the message for the `categoryItemClose` topic contains the index of the box that we want removed. Our publisher uses the `jQuery.fn.closest()` method to traverse the DOM and reach the child elements of our `boxContainer` element and then uses the `jQuery.fn.index()` method to find its position among its siblings. The subscriber then uses `jQuery.fn.eq()` and the `boxIndex` property from the received message to filter and remove only the requested information box from the dashboard.

In a more complex application, instead of the box index, we can associate each information box element with a newly retrieved `jQuery.guid` using a mapping object. This will allow our publisher to use that `guid` in the message instead of the (DOM-related) element index. The subscriber will then search the mapping object for that `guid` in order to locate and remove the appropriate box.

Since we are trying to demonstrate the advantages of the Pub/Sub Pattern, this implementation change was not introduced in order to ease the comparison with the Observer Pattern and is instead left as a recommended exercise for the reader.
To summarize the above, we used the `dashboardCategorySelect`, `categoryItemOpen`, and `categoryItemClose` topics as our application-level events in order to decouple the handling of the user actions from their origin (the UI element). As a result, we now have dedicated reusable pieces of code that manipulate our dashboard's content, which is equivalent to abstracting them into separate functions. This allows us to programmatically publish a series of messages so that we can, for example, remove all the existing information boxes and add all the category items of the currently selected category. Alternatively, even better, make the dashboard show all the items of each category for 10 seconds and then move to the next one.

**Extending the implementation**

In order to demonstrate the scalability that the Pub/Sub Pattern brings with it, we will extend our current example by adding a counter with the number of boxes that are currently open in the dashboard.

---

**Dashboard Example**

<table>
<thead>
<tr>
<th>Sales</th>
<th>1st week</th>
<th>2nd week</th>
<th>3rd week</th>
<th>4th week</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open boxes: 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product A</th>
<th>Advertisement 3</th>
<th>3rd week</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information box regarding Product A</td>
<td>Information box regarding Advertisement 3</td>
<td>Information box regarding 3rd week</td>
</tr>
</tbody>
</table>

For the counter implementation, we will need to add some extra HTML to our page and also create and reference a new JavaScript file to hold the counter implementation:

```html
...<section class="boxContainer">
</div>
</section>
<!-- ... -->
```

```html
</div>
</section>
```

```html
</div> <section class="boxContainer">
```
```
The Publish/Subscribe Pattern

In the HTML page of the example, we will need to add an extra <div> element to hold our counter and some description text. For our counter, we are using an <output> element, which is a semantic HTML5 element ideal to present results of user actions. The browser will use it just like a normal <span> element, so it will appear right next to its description. Also, since there is initially a hint box open in our dashboard, we use a 1 for its initial content:

```javascript
$(document).ready(function() {
  broker.on('categoryItemOpen categoryItemClose',
    function (event, message) {
      var $counter = $('#dashboardItemCounter');
      var count = parseInt($counter.text());

      if (event.type === 'categoryItemOpen') {
        $counter.text(count + 1);
      } else if (event.type === 'categoryItemClose' && count > 0) {
        $counter.text(count - 1);
      }
    });
});
```

For the counter implementation itself, all we need to do is add an extra subscriber to the dashboard's broker, which is globally available to other JavaScript files loaded in the page, since we have attached it to the window object. We are simultaneously subscribing to two topics, by passing them space delimited to the jQuery.fn.on() method. Right after this, we locate the counter <output> element that has the ID dashboardItemCounter and parse its text content as a number. In order to differentiate our action, based on the topic that the message has received, we use the event object that jQuery passes as the first parameter to our anonymous function, which is our subscriber. Specifically, we use the type property of the event object that holds the topic name of the message that was received and based on its value, we change the content of the counter.

For more information on the event object that jQuery provides, you can visit [http://api.jquery.com/category/events/event-object/](http://api.jquery.com/category/events/event-object/).

Similarly, we could also rewrite the code that prevents accidental double-clicks on the category item buttons. All that is needed is to add an extra subscriber for the categoryItemOpen topic and use the categoryName property of the message to locate the pressed button.
Using any object as a broker

While in our example we used the outermost container element of our dashboard for our broker, it is also common to use the $(document) object as a broker. Using the application's container element is considered a good semantic practice, which also scopes the emitted events.

As we described earlier in this chapter, jQuery actually allows us to use any object as a broker, even an empty one. As a result, we could instead use something such as window.broker = $({}); for our broker, in case we prefer it over using a page element.

By using newly constructed empty objects, we can also easily create several brokers, in case such a thing would be preferred for a specific implementation. Moreover, in case a centralized broker is not preferred, we could just make each publisher the broker of itself, leading to an implementation more like the first/basic variant of the Pub/Sub Pattern.

Since in most cases, a declared variable is used to access the application's broker within a page, there is little difference between the above approaches. Just choose the one that better matches your team's taste, and in case you change your mind at a later point, all you have to do is use a different assignment on your broker variable.

Using custom event namespaces

As a closing note for this chapter, we will present, in short, the mechanism that jQuery provides for namespacing custom events. The main benefit of event namespacing is that it allows us to use more specific event names that better describe their purpose, while also helping us to avoid conflicts between different implementation parts and plugins. It also provides a convenient way to unbind all the events of a given namespace from any target (element or broker).

A simple example implementation will look as follows:

```javascript
var broker = $({});
broker.on('close.dialog', function (event, message){
    console.log(event.type, event.namespace);
});
broker.trigger('close.dialog', ['messageEmitted']);
broker.off('.dialog');
// removes all event handlers of the "dialog" namespace
```
The Publish/Subscribe Pattern

For more information, you can visit the documentation page at http://docs.jquery.com/Namespaced_Events and the article at https://css-tricks.com/namespaced-events-jquery/ from the CSS-Tricks website.

Summary

In this chapter, we were introduced to the Publish/Subscribe Pattern. We saw its similarities with the Observer Pattern and also learned its benefits by doing a comparison of the two. We analyzed how the more distinct roles and the extra features that the Publish/Subscribe Pattern offers make it an ideal pattern for more complex use cases. We saw how jQuery developers adopted some of its concepts and brought them to their Observer Pattern implementation as custom events. Finally, we rewrote the example from the previous chapter using the Publish/Subscribe Pattern, adding some extra features and also achieving greater decoupling between the different parts and page elements of our application.

Now that we have completed our introduction to how the Publish/Subscribe Pattern can be used as a first step to decouple the different parts of an implementation, we can move on to the next chapter where we will be introduced to the Module Pattern. In the next chapter, we will learn how to separate the different parts of an implementation into independent modules and how to use namespace to achieve better code organization and define a strict API to achieve communication between the different modules.
Where to buy this book

You can buy jQuery Design Patterns from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.