Improve Your JavaScript Coding with Data-Oriented Programming

Use data binding to make your JavaScript coding more efficient

Dan Wahlin

Oct 18, 2012

Last month, in "Making the Move to Client-Side Development," I discussed several JavaScript data-binding frameworks and why they're essential as applications move more and more functionality to the client. Data binding is a key aspect of client-centric programming that can significantly minimize the amount of code written, simplify maintenance, and ultimately reduce the number of bugs that crop up in an application. Without data binding, you have to locate each control in a page with code and then assign or extract a value to or from it -- something I call "control-oriented" programming. By taking a more data-oriented approach, you enable an application's code to focus on the data, so that you don't need to worry nearly as much about the controls that are used.

In this article, I'll compare and contrast control-oriented JavaScript programming with data-oriented JavaScript programming and show a few examples of both types. Let's get started by taking a quick look at what control-oriented JavaScript programming looks like.

Control-Oriented Programming

What is control-oriented programming? Whenever an application uses explicit code to access a UI control by name or ID, control-oriented programming is being used to accomplish this. Listing 1 shows an example of control-oriented programming that uses jQuery.

```
function loadApprovalDiv()
{
    var subTotal = parseFloat($('#SubTotal').text());
    $('#ClientSubTotal').val(subTotal.toFixed(2));
    var salesTaxRate = parseFloat($('#SalesTaxRate').val()) / 100;
    var salesTaxAmount = (subTotal * salesTaxRate) * .9;
    var deliveryFee = parseFloat($('#DeliveryFee').val());
    var adminFee = ((subTotal + salesTaxAmount + deliveryFee) * .05);
    var total = (Round(subTotal) + Round(salesTaxAmount) + Round(deliveryFee) +
                 Round(adminFee));
    $('#ClientTotal').val(total);
    var deliveryAddress = $('#Delivery_Street').val();
    //See if they entered a suite
    if ($('#Delivery_Suite').val() != '')
      deliveryAddress += ', Suite ' + $('#Delivery_Suite').val();
    deliveryAddress += ' ' + $('#Delivery_City').val() + ' ' +
                       $('#Delivery_StateID option:selected').text() + ' ' +
                       $('#Delivery_Zip').val();

    var data = {
                  FinalSubTotal  : subTotal.toFixed(2),
                  FinalSalesTax  : salesTaxAmount.toFixed(2),
                  FinalTotal     : total.toFixed(2),
                  DeliveryFee    : deliveryFee.toFixed(2),
                  AdminFee       : adminFee.toFixed(2),
                  DeliveryName   : $('#Delivery_Name').val(),
                  DeliveryAddress: deliveryAddress,
                  DeliveryDate   : $('#Delivery_DeliveryDate').val(),
                  DeliveryTime   : $('#Delivery_DeliveryTime ' +
                                     'option:selected').text(),
                  MainItems      : generateJson('Main'),
                  AccessoryItems : generateJson('Accessory')
              };


        $("#OrderSummaryOutput").html(
            $("#OrderSummaryTemplate").render(data)
        );
}
```

As you look through the code in Listing 1, you can see that much of it is dedicated to finding controls in the page and extracting their values. This works absolutely fine -- after all, many applications take this approach. However, when an application is focused on controls and not on data, a lot of extra code and plumbing ends up being written, which complicates things if control IDs are changed, new controls are added, or existing controls are removed.

If you have only a few controls, the extra code isn't a big deal, but as the number of controls grows, writing and maintaining this code becomes a challenge. I think the cheese is definitely moving when it comes to client-side programming and that the smart money is on building data-oriented applications rather than control-oriented applications like the example I just described. That's why we're seeing the release of more and more data-binding frameworks for JavaScript.

Does this mean that frameworks such as jQuery, Zepto, and others will no longer be used? I think these types of selector frameworks still have their place and play a key role. I've been a huge fan of jQuery for many years and still use it in every project for various purposes, from finding elements to animating to working with plug-ins. However, I do find myself using jQuery less for finding controls and getting data in and out of them. For those types of tasks, I prefer to take a more data-oriented approach.

Data-Oriented Programming

By using a data-binding library or framework such as Knockout or AngularJS, you can wire JavaScript object properties to controls and have the controls and object properties update automatically (a process referred to as "two-way" binding) as changes are made on either side without having to write control-specific code. This means that you don't have to write selectors to find controls in the Document Object Model (DOM) and update them or grab values. If you've ever worked with application frameworks such as Adobe Flex, Silverlight, or Windows Presentation Foundation (WPF), you're used to this type of functionality, and I'm willing to bet that you can't live without it. Data binding is addictive once you start using it.

With application frameworks like Flex or Silverlight, the data-binding engine is built in, so you use whatever the framework gives you. The challenge in the JavaScript world is that there isn't simply one "best" data-binding framework to choose. Many different script libraries and frameworks are appearing on the scene, each with its own set of pros and cons.

I refer to applications that use data binding as being "data-oriented" because they're focused on the actual data as opposed to containing code to access controls in a given page ("control-oriented," as mentioned earlier). I've built many control-oriented applications over the years and found that making the transition to building data-oriented applications requires a different thought process. However, in my experience, making the move to building data-oriented applications is well worth the effort and ultimately results in better applications. I think making this transition is especially important for client-centric applications built using JavaScript.

Although I'm a big fan of jQuery, in recent years I've started realizing that when jQuery is used mainly to build control-oriented applications (where jQuery is used to find controls, update values, extract values, and perform similar actions), much of the code written in these apps is unnecessary and could be eliminated by using a data-oriented framework. jQuery absolutely has its place in applications, but using it to build control-oriented applications isn't a good use of its functionality, in my opinion, given some of the other options now available. jQuery is great when you require low-level DOM access but not as great when an application has a lot of create, read, update, and delete (CRUD) operations going on. This probably sounds a bit controversial given jQuery's popularity (and to people who know I'm a huge jQuery fan), but when you understand what a data-oriented application is and why it's important, then using a data-oriented framework makes more sense for many CRUD applications.

What does a data-oriented approach look like? Listing 2 shows a simple example of data binding in HTML with a JavaScript library called Knockout. This is the Knockout "Hello World" example.

```
<p>First name: <input data-bind="value: firstName" /></p>
<p>Last name: <input data-bind="value: lastName" /></p>
<h2>Hello, <span data-bind="text: fullName"> </span>!</h2>
```

As you look through the code in Listing 2, one of the first things you'll probably notice is that the elements in the view don't have any IDs on them (although IDs could certainly be added). Instead, a data-bind attribute is used to bind the input element values to properties (firstName and lastName, respectively), and the span's text value is bound to fullName. As the data changes in the data source (officially called a ViewModel), the input and span elements will automatically be updated. Likewise, as a user changes the information in the input elements, the ViewModel will automatically be changed. Listing 3 shows what the ViewModel looks like.

```
var ViewModel = function(first, last) {
    this.firstName = ko.observable(first);
    this.lastName = ko.observable(last);

    this.fullName = ko.computed(function() {
            return this.firstName() + " " + this.lastName();
    }, this);
};

ko.applyBindings(new ViewModel("Planet", "Earth"));
```

Using a data-oriented approach cleans up the code significantly and also lets you write tests using frameworks such as QUnit without having to involve the UI and any associated controls in it. Separation of the data and rules from the UI lets you write more maintainable code. You can find a more detailed example of using a data-oriented approach on the Knockout site.

More to Come

I've discussed the differences between control-oriented programming and data-oriented programming in client-centric applications. We'll continue the discussion about data-oriented JavaScript programming in my next article, when I'll jump into AngularJS, one of my favorite data-binding frameworks, and show how it can be used to build data-oriented JavaScript applications.

Learn More About JavaScript Programming

- Calling a Server-Side Page Method from Client-Side JavaScript
- Explore the New World of JavaScript-Focused Client-Side Web Development
- JavaScript Closures
- A Deeper Look at JavaScript Closures
- Localize Your Client-Side JavaScript Applications
- Using MVVM in JavaScript with Knockout

Related Articles

Knocking Myself Out for Avoiding KnockoutJS

Rendering Data on the Client with jQuery Templates

How to Build a jQuery HTML5 Web Application with Client-Side Coding

4 More Web Development Tips to Improve Your jQuery Coding 2

5 Web Development Tips to Improve Your jQuery Coding