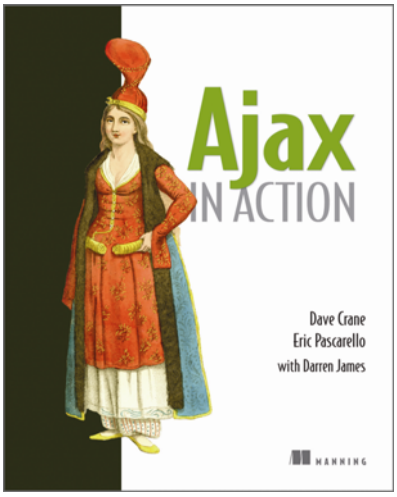
Excerpt

The following is the third and final installment of Ajax in Action chapter 4 from Manning Publications



Smarter MVC: Using the Model to generate the View

By David Crane and Eric Pascarello
with Darren James

Ajax applications can contain much more client-side code than a standard web application, and hence benefit much more from the order that patterns and refactoring bring. Chapter 4 is the first of three chapters that apply refactoring and patterns to the client-side codebase. You won't see much of the asynchronous requests that give Ajax its name in this chapter, but the style of programming that we're discussing here is a direct consequence of being able to make asynchronous requests.

Dividing the codebase into three discrete layers using MVC makes the JavaScript code in an Ajax application more ordered and maintainable, but it also results in a certain amount of overhead of communication between the tiers. In this installation, we look at a way of reducing this overhead by generating the View layer automatically from the Model.

4.5 Generating the View from the Model

By introducing MVC into the browser, we've given ourselves three distinct subsystems to worry about. Separating concerns may result in cleaner code, but it can also result in a lot of code, and a common critique of design patterns is that they can turn even the simplest task into quite an involved process (as Enterprise JavaBeans [EJB] developers know only too well!).

Many-layered application designs often end up repeating information across several layers. We know the importance of DRY code, and a common way of tackling this repetition is to define the necessary information once, and generate the various layers automatically from that definition. In this section, we'll do just that, and present a technique that simplifies the MVC implementation and brings together all three tiers in a simple way. Specifically, we'll target the View layer.

So far, we've looked at the View as a hand-coded representation of the underlying Model. This gives us considerable flexibility in determining what the user sees, but at times, we won't need this flexibility, and hand-coding the UI can become tedious and repetitive. An alternative approach is to automatically generate the user interface, or at least portions of it, from the underlying Model. There are precedents for doing this, such as the Smalltalk language environments and the Java/.NET Naked Objects framework (see the Resources section), and JavaScript is well suited to this sort of task. Let's have a look at what JavaScript reflection can do for us in this regard, and develop a generic "Object Browser" component, that can be used as a View for any JavaScript object that we throw at it.

4.5.1 Reflecting on a JavaScript object

Most of the time when we write code to manipulate an object, we already have a fairly good idea of what the object is and what it can do. Sometimes, however, we need to code blindly, as it were, and examine the object without any prior knowledge. Generating a user interface for our domain model objects is just such a case. Ideally, we would like to develop a reusable solution that can be equally applied to any domain—finance, e-commerce, scientific visualization, and so on. This section presents just such a JavaScript library, the ObjectViewer, that can be used in your own applications. To give you a taste of the ObjectViewer in action, figure 4.7 shows the ObjectViewer displaying several layers of a complex object graph.

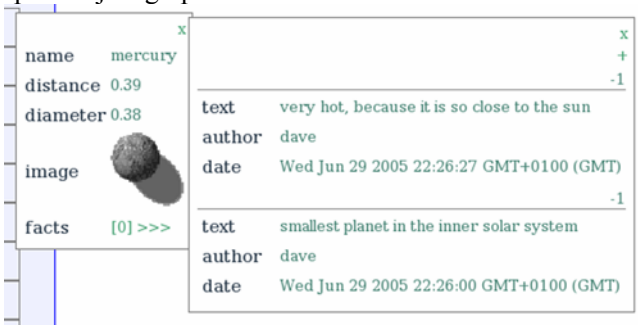


Figure 4.7 Here the ObjectViewer is used to display a hierarchical system of planets, each of which contains a number of informational properties, plus a list of facts stored as an array.

The object being viewed, representing the planet Mercury, is quite sophisticated, with properties including an image URL, an array of facts, as well as simple strings and numbers. Our ObjectViewer can handle all of these intelligently without knowing anything specific about the type of object in advance.

The process of examining an object and querying its properties and capabilities is known as *reflection*. Readers with a familiarity to Java or .NET should already be familiar with this term. We discuss JavaScript's reflection capabilities in more detail in appendix B. To summarize briefly here, a JavaScript object can be iterated over as if it were an associative array. To print out all the properties of an object, we can simply write

```
var description="";
for (var i in MyObj){
  var property=MyObj[i];
  description+=i+" = "+property+"\n";
}
alert(description);
```

Presenting data as an alert is fairly primitive and doesn't integrate with the rest of a UI very well. Listing 4.11 presents the core code for the ObjectViewer object.

Listing 4.11 ObjectViewer object

```
objviewer.ObjectViewer=function(obj,div,isInline,addNew){
  styling.removeAllChildren(div);
  this.object=obj;
  this.mainDiv=div;
  this.mainDiv.viewer=this;
  this.isInline=isInline;
  this.addNew=addNew;
  var table=document.createElement("table");
  this.tbod=document.createElement("tbody");
  table.appendChild(this.tbod);
  this.fields=new Array();
  this.children=new Array();
  for (var i in this.object){
    this.fields[i]=new objviewer.PropertyViewer(
      this, i
    );
  }
objviewer.PropertyViewer=function(objectViewer,name){
  this.objectViewer=objectViewer;
  this.name=name;
  this.value=objectViewer.object[this.name];
```

```
  this.rowTr=document.createElement("tr");
  this.rowTr.className='objViewRow';
  this.valTd=document.createElement("td");
  this.valTd.className='objViewValue';
  this.valTd.viewer=this;
  this.rowTr.appendChild(this.valTd);
  var valDiv=this.renderSimple();
  this.valTd.appendChild(valDiv);
  viewer.tbod.appendChild(this.rowTr);
}
objviewer.PropertyViewer.prototype.renderSimple=function(){
    var valDiv=document.createElement("div");
    var valTxt=document.createTextNode(this.value);
    valDiv.appendChild(valTxt);
    if (this.spec.editable){
      valDiv.className+=" editable";
      valDiv.viewer=this;
      valDiv.onclick=objviewer.PropertyViewer.editSimpleProperty;
    }
    return valDiv;
}
```

Our library contains two objects: an ObjectViewer, which iterates over the members of an object and assembles an HTML table in which to display the data, and a PropertyViewer, which renders an individual property name and value as a table row.

This gets the basic job done, but it suffers from several problems. First, it will iterate over every property. If we have added helper functions to the Object prototype, we will see them. If we do it to a DOM node, we see all the built-in properties and appreciate how heavyweight a DOM element really is. In general, we want to be selective about which properties of our object we show to the user. We can specify which properties we want to display for a given object by attaching a special property, an Array, to the object before passing it to the object renderer. Listing 4.12 illustrates this.

Listing 4.12 Using the `objViewSpec` property

```
objviewer.ObjectViewer=function(obj,div,isInline,addNew){
  styling.removeAllChildren(div);
  this.object=obj;
  this.spec=objviewer.getSpec(obj);
  this.mainDiv=div;
  this.mainDiv.viewer=this;
  this.isInline=isInline;
  this.addNew=addNew;
  var table=document.createElement("table");
```

```
  this.tbod=document.createElement("tbody");
  table.appendChild(this.tbod);
  this.fields=new Array();
  this.children=new Array();
  for (var i=0;i<this.spec.length;i++){
    this.fields[i]=new objviewer.PropertyViewer(
      this,this.spec[i]
    );
  }
objviewer.getSpec=function (obj){
  return (obj.objViewSpec) ?
    obj.objViewSpec :
    objviewer.autoSpec(obj);
}
objviewer.autoSpec=function(obj){
  var members=new Array();
  for (var propName in obj){
    var spec={name:propName};
    members.append(spec);
  }
  return members;
}
objviewer.PropertyViewer=function(objectViewer,memberSpec){
  this.objectViewer=objectViewer;
  this.spec=memberSpec;
  this.name=this.spec.name;
  ...
}
```

We define a property `objViewSpec`, which the ObjectViewer constructor looks for in each object. If it can't find such a property, it then resorts to creating one by iterating over the object in the `autoSpec()` function. The `objViewSpec` property is a numerical array, with each element being a lookup table of properties. For now, we're only concerned with generating the `name` property. The PropertyViewer is passed the spec for this property in its constructor and can take hints from the spec as to how it should render itself.

If we provide a specification property to an object that we want to inspect in the ObjectViewer, then we can limit the properties being displayed to those that we think are relevant.

A second problem with our ObjectViewer is that it doesn't handle complex properties very well. When objects, arrays, and functions are appended to a string, the `toString()` method is called. In the case of an object, this generally returns something nondescriptive such as `[Object object]`. In the case of a Function object, the entire source code for the function is returned. We need to discriminate between the different types of properties, which we can do

using the `instanceof` operator. With that in place, let's see how we can improve on our viewer.

4.5.2 Dealing with arrays and objects

One way of handling arrays and objects is to allow the user to drill down into them using separate ObjectViewer objects for each property. There are several ways of representing this. We have chosen here to represent child objects as pop-out windows, somewhat like a hierarchical menu.

To achieve this, we need to do two things. First, we need to add a `type` property to the object specification and define the types that we support:

```
objviewer.TYPE_SIMPLE="simple";
objviewer.TYPE_ARRAY="array";
objviewer.TYPE_FUNCTION="function";
objviewer.TYPE_IMAGE_URL="image url";
objviewer.TYPE_OBJECT="object";
```

We modify the function that generates specs for objects that don't come with their own to take account of the type, as shown in listing 4.13.

Listing 4.13 Modified `autoSpec()` function

```
objviewer.autoSpec=function(obj){
  var members=new Array();
  for (var propName in obj){
    var propValue=obj[name];
    var propType=objviewer.autoType(value);
    var spec={name:propName,type:propType};
    members.append(spec);
  }
  if (obj && obj.length>0){
    for(var i=0;i<obj.length;i++){
      var propName="array ["+i+"]";
      var propValue=obj[i];
      var propType=objviewer.ObjectViewer.autoType(value);
      var spec={name:propName,type:propType};
      members.append(spec);
    }
  }
  return members;
}
objviewer.autoType=function(value){
  var type=objviewer.TYPE_SIMPLE;
  if ((value instanceof Array)){
```

```
    type=objviewer.TYPE_ARRAY;
  }else if (value instanceof Function){
    type=objviewer.TYPE_FUNCTION;
  }else if (value instanceof Object){
    type=objviewer.TYPE_OBJECT;
  }
  return type;
}
```

Note that we also add support for numerically indexed arrays, whose elements wouldn't be discovered by the `for...in` style of loop.

The second thing that we need to do is to modify the PropertyViewer to take account of the different types and render them accordingly, as shown in listing 4.14.

Listing 4.14 Modified PropertyViewer constructor

```
objviewer.PropertyViewer=function
 (objectViewer,memberSpec,appendAtTop){
  this.objectViewer=objectViewer;
  this.spec=memberSpec;
  this.name=this.spec.name;
  this.type=this.spec.type;
  this.value=objectViewer.object[this.name];
  this.rowTr=document.createElement("tr");
  this.rowTr.className='objViewRow';
  var isComplexType=(this.type==objviewer.TYPE_ARRAY
                    ||this.type==objviewer.TYPE_OBJECT);
  if ( !(isComplexType && this.objectViewer.isInline
)
  ){
    this.nameTd=this.renderSideHeader();
    this.rowTr.appendChild(this.nameTd);
  }
  this.valTd=document.createElement("td");
  this.valTd.className='objViewValue';
  this.valTd.viewer=this;
  this.rowTr.appendChild(this.valTd);
  if (isComplexType){
    if (this.viewer.isInline){
      this.valTd.colSpan=2;
      var nameDiv=this.renderTopHeader();
      this.valTd.appendChild(nameDiv);
      var valDiv=this.renderInlineObject();
      this.valTd.appendChild(valDiv);
    }else{
      var valDiv=this.renderPopoutObject();
```

```
      this.valTd.appendChild(valDiv);
    }
  }else if (this.type==objviewer.TYPE_IMAGE_URL){
    var valImg=this.renderImage();
    this.valTd.appendChild(valImg);
  }else if (this.type==objviewer.TYPE_SIMPLE){
    var valTxt=this.renderSimple();
    this.valTd.appendChild(valTxt);
  }
  if (appendAtTop){
    styling.insertAtTop(viewer.tbod,this.rowTr);
  }else{
    viewer.tbod.appendChild(this.rowTr);
  }
}
```

To accommodate the various types of properties, we have defined a number of rendering methods, the implementation of which is too detailed to reproduce in full here. Source code for the entire ObjectViewer can be downloaded from the website that accompanies this book.

We now have a fairly complete way of viewing our domain model automatically. To make the domain model objects visible, all that we need to do is to assign `objViewSpec` properties to their prototypes. The Planet object backing the view shown in figure 4.7, for example, has the following statement in the constructor:

```
this.objViewSpec=[
  {name:"name",     type:"simple"},
  {name:"distance", type:"simple", editable:true},
  {name:"diameter", type:"simple", editable:true},
  {name:"image",    type:"image url"},
  {name:"facts",    type:"array",  addNew:this.newFact, inline:true }
];
```

The notation for this specification is the JavaScript object notation, known as JSON. Square braces indicate a numerical array, and curly braces an associative array or object (the two are really the same). We discuss JSON more fully in appendix B.

There are a few unexplained entries here. What do `addNew`, `inline`, and `editable` mean? Their purpose is to notify the View that these parts of the domain model can not only be inspected but also modified by the user, bringing in the Controller aspects of our system, too. We'll look at this in the next section.

4.5.3 Adding a Controller

It's nice to be able to look at a domain model, but many everyday applications require us to modify them too—download the tune, edit the document, add items to the shopping basket,

and so on. Mediating between user interactions and the domain model is the responsibility of the Controller, and we'll now add that functionality to our ObjectViewer.

The first thing that we'd like to do is to be able to edit simple text values when we click on them, if our specification object flags them as being editable. Listing 4.15 shows the code used to render a simple text property.

Listing 4.15 `renderSimple()` function

```
objviewer.PropertyViewer.prototype.renderSimple=function(){
    var valDiv=document.createElement("div");
    var valTxt=document
      .createTextNode(this.value);
    valDiv.appendChild(valTxt);
    if (this.spec.editable){
      valDiv.className+=" editable";
      valDiv.viewer=this;
      valDiv.onclick=objviewer.PropertyViewer.editSimpleProperty;
    }
    return valDiv;
}

objviewer.PropertyViewer.editSimpleProperty=function(e){
  var viewer=this.viewer;
  if (viewer){
    viewer.edit();
  }
}
objviewer.PropertyViewer.prototype.edit=function(){
  if (this.type=objviewer.TYPE_SIMPLE){
    var editor=document.createElement("input");
    editor.value=this.value;
    document.body.appendChild(editor);
    var td=this.valTd;
    xLeft(editor,xLeft(td));
    xTop(editor,xTop(td));
    xWidth(editor,xWidth(td));
    xHeight(editor,xHeight(td));
    td.replaceChild(editor,td.firstChild);
    editor.onblur=objviewer.
      PropertyViewer.editBlur;
    editor.viewer=this;
```

```
    editor.focus();
  }
}
objviewer.PropertyViewer
                          ❺ Finish editing
 .editBlur=function(e){
  var viewer=this.viewer;
  if (viewer){
    viewer.commitEdit(this.value);
  }
}
objviewer.PropertyViewer.prototype.commitEdit=function(value){
  if (this.type==objviewer.TYPE_SIMPLE){
    this.value=value;
    var valDiv=this.renderSimple();
    var td=this.valTd;
    td.replaceChild(valDiv,td.firstChild);
    this.objectViewer
                          ❻ Notify observers
     .notifyChange(this);
  }
}
```

Editing a property involves several steps. First, we want to assign an `onclick` handler to the DOM element displaying the value, if the field is editable ❶. We also assign a specific CSS classname to editable fields, which will make them change color when the mouse hovers over them. We need the user to be able to realize that she can edit the field, after all.

`editSimpleProperty()` ❷ is a simple event handler that retrieves the reference to the PropertyViewer from the clicked DOM node and calls the `edit()` method. This way of connecting the View and Controller should be familiar from section 4.3.1. We check that the property type is correct and then replace the read-only label with an equivalent-sized HTML form text input, containing the value ❸. We also attach an `onblur` handler to this text area ❹, which replaces the editable area with a read-only label ❺ and updates the domain model.

We can manipulate the domain model in this way, but in general, we would often like to take some other action when the model is updated. The `notifyChange()` method of the ObjectViewer ❻, invoked in the `commitEdit()` function, comes into play here. Listing 4.16 shows this function in full.

Listing 4.16 `ObjectViewer.notifyChange()`

```
objviewer.ObjectViewer.prototype
 .notifyChange=function(propViewer){
  if (this.onchangeRouter){
    this.onchangeRouter.notify(propViewer);
  }
  if (this.parentObjViewer){
    this.parentObjViewer.notifyChange(propViewer);
  }
}
objviewer.ObjectViewer.prototype
 .addChangeListener=function(lsnr){
  if (!this.onchangeRouter){
    this.onchangeRouter=new jsEvent.EventRouter(this,"onchange");
  }
  this.onchangeRouter.addListener(lsnr);
}
objviewer.ObjectViewer.prototype
 .removeChangeListener=function(lsnr){
  if (this.onchangeRouter){
    this.onchangeRouter.removeListener(lsnr);
  }
}
```

The problem we are facing—notifying arbitrary processes of a change in our domain model—is ideally solved by the Observer pattern and the EventRouter object that we defined in section 4.3.3. We could attach an EventRouter to the `onblur` event of the editable fields, but a complex model may contain many of these, and our code shouldn't have visibility of such fine details in the ObjectViewer implementation.

Instead, we define our own event type on the ObjectViewer itself, an `onchange` event, and attach an EventRouter to that. Because our ObjectViewers are arranged in a tree structure when drilling down on object and array properties, we pass `onchange` events to the parent, recursively. Thus, in general, we can attach listeners to the root ObjectViewer, the one that we create in our application code, and changes to model properties several layers down the object graph will propagate back up to us.

A simple example of an event handler would be to write a message to the browser status bar. The top-level object in a model of planets is the solar system, so we can write

```
var topview=new objviewer.ObjectViewer
  (planets.solarSystem,mainDiv);
topview.addChangeListener(testListener);
```

where `testListener` is an event-handler function that looks like this:

```
function testListener(propviewer){
  window.status=propviewer.name+" ["+propviewer.type+"] = "+propviewer.value;
}
```

Of course, in reality, we would want to do more exciting things when the domain model changes, such as contacting the server. In the next chapter, we'll look at ways of contacting the server and put our ObjectViewer to further use.

4.6 Summary

The Model-View-Controller pattern is an architectural pattern that has been applied to the server code of classic web applications. We showed how to reuse this pattern on the server in an Ajax application, in order to generate data feeds for the client. We also applied the pattern to the design of the client itself and developed a range of useful insights through doing so.

Looking at the View subsystem, we demonstrated how to effectively separate presentation from logic, with the very practical benefit of allowing designer and programmer roles to be kept separate. Maintaining clear lines of responsibilities in the codebase that reflect your team's organizational structure and skill sets can be a great productivity booster.

In the Controller code, we looked at the different event models available to Ajax and erred on the side of caution toward the older event model. Although it is limited to a single callback function for each event type, we saw how to implement the Observer pattern to develop a flexible, reconfigurable event-handler layer on top of the standard JavaScript event model.

Regarding the Model, we began to address the larger issues of distributed multiuser applications, which we will explore further in chapter 5.

Looking after a Model, a View, and a Controller can seem like a lot of work. In our discussion of the ObjectViewer example, we looked at ways of simplifying the interactions between these using automation, and we created a simple system capable of presenting an object model to the user and allowing interaction with it.

We'll continue to draw upon design patterns as we move on to explore client/server interactions in the next chapter.