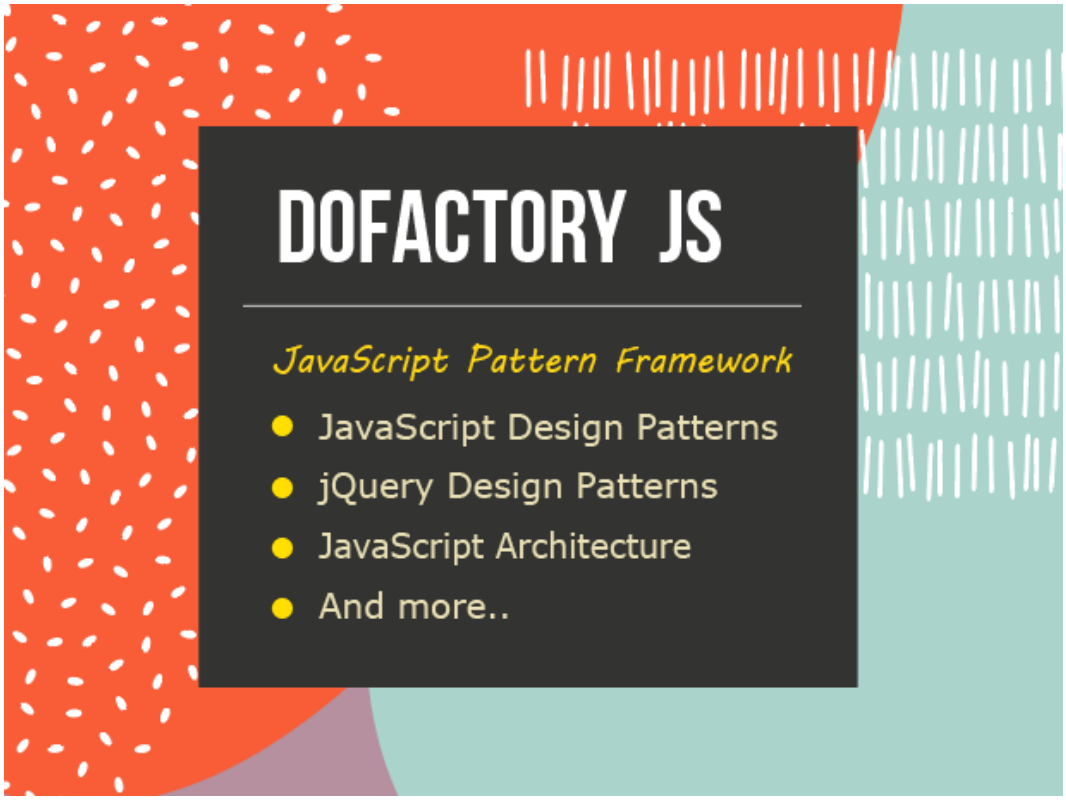
Dofactory JS 6.0

JavaScript Patterns in Action



by

Data & Object Factory, LLC

www.dofactory.com

8. JavaScript Patterns in Action

Index

Introduction

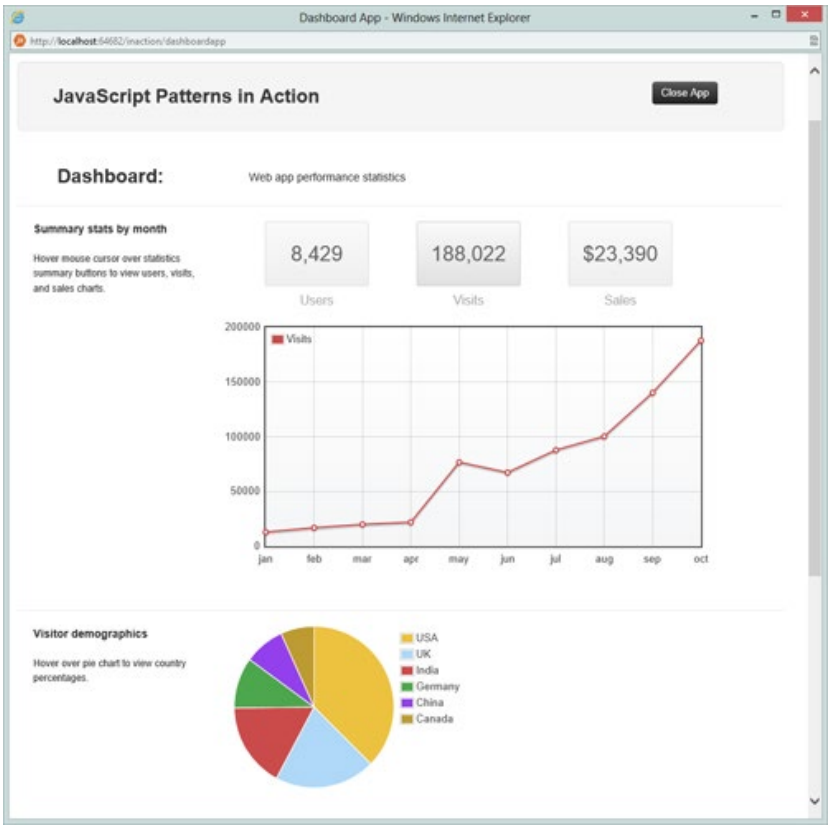
The goal of this section is to demonstrate Idioms and Design Patterns in a more realistic context. These pages are client only; any Ajax access to the server has been *mocked* with a tool called *fauxServer*. Several pages make use of the popular *Backbone* MVC Framework. The charting is implemented with the *flot* library.

There are many things you can build with JavaScript. The Patterns in Action examples in this section represent a sampling of the more common uses in web app development: Dashboard (charting), record management with CRUD (create, read, update, and delete), search, and pagination. Most apps today feature one or more of these items. Feel free to use these examples as a starting point in your own projects.

When reviewing the code, you'll see that numerous Idioms and Design Patterns have been used. Special `// **` comments (with two asterisks) are included to make it easy to locate these patterns. A final note: we did not set out to cram as many patterns into a single page; what you see here is fairly representative of real-world, pattern-based application code.

Dashboard

Here we have a dashboard app displaying dynamic line and pie charts. First off, here is a screenshot of the app:



The dashboard app contains charts and summary statistics on a hypothetical website. At the top is a line chart that displays users/month, visits/month, and sales/month. You change the charts by hovering the mouse cursor over one of the three large buttons. The buttons themselves display the current monthly values.

Next is a pie chart which shows the distribution of the website's demographics. Most visitors are coming from the US, then UK, India, Germany, etc. Hover the mouse over a slice to get the country and percentages for that slice.

At the bottom you find a real-time chart showing the number of concurrent sessions. They are updated in real-time. Of course, these figures are simulated.

To display the charts we use *flot* which an open source JavaScript charting package written by Google. It requires HTML5. In case your browser does not support the *canvas* element a message will suggest that you upgrade to a more recent browser version. All recent browsers today do support canvas.

The JavaScript is listed below (the code is fairly lengthy; it may be best to open the actual source code in a separate editor and read along).

```
var Patterns = {
    // ** namespace pattern
    namespace: function (name) {

        // ** single var pattern
        var parts = name.split(".");
        var ns = this;

        // ** iterator pattern
        for (var i = 0, len = parts.length; i < len; i++) {
            // ** || idiom
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

// ** namespace pattern
// ** revealing module pattern
// ** singleton pattern
Patterns.namespace("InAction").Dashboard = (function () {

    // ** immediate function idiom
    var users = (function () {
        // ** lazy load pattern (using closure)
        var data;
        var show = function () {
            // ** truthy/falsy idiom
            // ** || idiom
            data = data || get("users");
            showPlot(data);
        }
        return { show: show };
    })();

    // ** immediate function idiom
    var visits = (function () {
        // ** lazy load pattern (using closure)
        var data;
```

```
        var show = function () {
            // ** truthy/falsy idiom
            // ** || idiom
            data = data || get("visits");
            showPlot(data);
        }
        return { show: show };
    })();

    // ** immediate function idiom
    var sales = (function () {
        // ** lazy load pattern (using closure)
        var data;
        var show = function () {
            // ** truthy/falsy idiom
            // ** || idiom
            data = data || get("sales");
            showPlot(data);
        }
        return { show: show };
    })();

    // ** strategy pattern
    var strategy = users;  // default strategy

    var showPlot = function (data) {

        // ** option hash idiom
        var options = {
            legend: { position: "nw" },
            lines: { show: true },
            points: { show: true },
            xaxis: { ticks: [[1, "jan"], [2, "feb"], [3, "mar"], [4, "apr"],
                             [5, "may"], [6, "jun"], [7, "jul"], [8, "aug"],
                             [9, "sep"], [10, "oct"], [11, "nov"], [12, "dec"]] },
            grid: { backgroundColor: { colors: ["#fff", "#f5f6f7"] } }
        };

        $.plot($("#plotarea"), data, options);
    }

    var pieHover = function (event, pos, obj) {

        // ** truthy/falsy idiom
        // ** && idiom
        if (obj && obj.series) {
```

```
            percent = parseFloat(obj.series.percent).toFixed(2);
            $("#hover").html('<span style="font-weight: bold;">' +
               obj.series.label + ' (' + percent + '%)</span>');
        }
    }

    var pieClick = function (event, pos, obj) {

        // ** truthy/falsy idiom
        // ** && idiom
        if (obj && obj.series) {
            percent = parseFloat(obj.series.percent).toFixed(2);
            alert('' + obj.series.label + ': ' + percent + '%');
        }
    }

    var initPie = function () {
        var piedata = [
                { label: "USA", data: 110 },
                { label: "UK", data: 60 },
                { label: "India", data: 50 },
                { label: "Germany", data: 30 },
                { label: "China", data: 24 },
                { label: "Canada", data: 20 }
        ];

        var $piearea = $("#piearea");

        // ** option hash idiom
        $.plot($piearea, piedata,
            {
                series: {
                    pie: {
                        show: true
                    }
                },
                grid: {
                    hoverable: true,
                    clickable: true
                }
            });

        // ** chaining pattern
        $piearea.bind("plothover", pieHover)
                .bind("plotclick", pieClick)
                .hover(function () { $("#hover").css("visibility", "visible"); },
```

```
                    function () { $("#hover").css("visibility", "hidden"); }
        );
    };

    var showLineChart = function () {
        strategy.show();
    }

    var initPlots = function () {

        // ** observer pattern
        $("#link-users").on('hover', function () {
            $('[id^="link-"]').removeClass("active");
            $("#link-users").addClass("active");
            // ** strategy pattern
            strategy = users;
            showLineChart();
        });
        $("#link-visits").on('hover', function () {
            $('[id^="link-"]').removeClass("active");
            $("#link-visits").addClass("active");
            // ** strategy pattern
            strategy = visits;
            showLineChart();
        });
        $("#link-sales").on('hover', function () {
            $('[id^="link-"]').removeClass("active");
            $("#link-sales").addClass("active");
            // ** strategy pattern
            strategy = sales;
            showLineChart();
        });

        showLineChart();
    };

    var initRealtime = function () {

        var data = [];

        function getData() {
            if (data.length > 0) data = data.slice(1);

            // generate random data
            while (data.length < 300) {
```

```
                var prev = data.length > 0 ? data[data.length - 1] : 50;
                var y = Math.round(prev + Math.random() * 5 - 2.5);

                y = Math.max(Math.min(y, 100), 0); // range is 0 - 100;
                data.push(y);
            }

            // return x and y coordinate pairs
            var coordinates = [];
            for (var x = 0; x < data.length; ++x) {
                coordinates.push([x, data[x]])
            }
            return coordinates;
        }

        // ** option hash idiom
        var options = {
            series: { shadowSize: 0 },
            yaxis: { min: 0, max: 100 },
            xaxis: { show: false },
            colors: ["#cc0000"]
        };
        var plot = $.plot($("#realtime"), [getData()], options);

        function updateRealtime() {

            plot.setData([getData()]);
            plot.draw();   // call draw because axis did not change

            // ** zero timeout pattern  (30 milliseconds)
            setTimeout(updateRealtime, 30);
        }

        updateRealtime();
    };

    // mock server call
    var get = function (what) {
        switch(what) {
            case "users": return [{ "label": "Users", "data": [[1, 344], [2, 578], [3,
460], [4, 902], [5, 1933], [6, 2303], [7, 3281], [8, 3590], [9, 6830], [10, 8429]] }];
            case "visits": return [{ "label": "Visits", "data": [[1, 12965], [2,
16935], [3, 19993], [4, 21983], [5, 76801], [6, 67372], [7, 87922], [8, 100399], [9,
140332], [10, 188022]], "color": 2 }];
```

```
            case "sales": return [{ "label": "Sales", "data": [[1, 266], [2, 1009],
[3, 6754], [4, 6570], [5, 7489], [6, 8888], [7, 10821], [8, 14099], [9, 12222], [10,
23390]], "color": 3 }];
            default: return [];
        }
    }

    var start = function () {
        initPlots();
        initPie();
        initRealtime();
    };

    return { start: start };
})();


$(function () {

    // First check for HTML5 canvas support

    // ** double !! idiom
    var supportsCanvas = !!document.createElement("canvas").getContext;
    if (!supportsCanvas) {
        // ** chained pattern
        $("#message").css("color", "red").html("It seems that your browser does not
support HTML canvas. Please upgrade to a more recent version.");
        return;
    }

    // ** facade pattern
    var dashboard = Patterns.InAction.Dashboard;
    dashboard.start();
});
```

The following patterns and idioms are present in this code:

- Immediate function idiom
- Double !! idiom
- Truthy/falsy idiom
- || and && idiom
- Option Hash idiom
- Namespace pattern

- Single var pattern
- Module pattern
- Chaining pattern
- Lazy Load pattern
- Zero Timeout pattern
- Iterator pattern
- Singleton pattern
- Strategy pattern
- Observer pattern
- Façade pattern

It is interesting to note the large number of patterns and idioms in only 250 lines of code. As mentioned earlier we did not set out to get as many patterns as possible into the script. However, what this demonstrates is how pervasive patterns and idioms are in today's JavaScript.

The Namespace pattern allows us to keep the entire code base in the `Patterns.InAction` namespace. The `Dashboard` module was built with the Module pattern, more specifically the Revealing Module pattern which exposes only a single method: the `start` method. Everything else in `Dashboard` is hidden from the outside world. Only a single `Dashboard` instance can exist which is the idea behind the Singleton pattern.

The `Dashboard` module contains 3 'subsystems': line charts, pie chart, and real-time chart. Each one uses numerous private variables and functions. This represents the Façade pattern which exposes a simple API and hides the complexity of a set of subsystems. This API is extremely simple because the only way to access the Façade is through the `start` method.

Each one of the three line-charts at the top has its own function object; they are named `users`, `visits`, and `sales` and are constructed using the Immediate Function idiom. Each one only exposes a single method named `show`. The data is lazy loaded and kept inside of the function objects using the function's closures. This is the Lazy Load pattern in action – Lazy Initialization to be exact.

The line `data = data || get("users");` is built around two idioms: the truthy/falsy idiom and the || idiom. First it checks whether the data variable is truthy: if so, we shortcut the ||

and assign data to itself. If not, the data is retrieved using `get` which is a mocked server call. This statement also gets to the heart of the Lazy Load pattern because data is only loaded when absolutely necessary. Once loaded we will not load it again.

The `users`, `visits`, and `sales` function objects are used in a Strategy Pattern. The `strategy` variable is bound to the line-chart that is currently displayed. If a different chart needs be displayed (because the user moves the mouse cursor over a different button), the `strategy` gets reassigned with a different strategy function object. The `showLineChart` function invokes the strategy's `show` method without really knowing which strategy is used.

The Option Hash idiom is used to configure the chart options. The *flot* package makes extensive use of this idiom.

The Chaining pattern lets us chain multiple jQuery event bindings to the Pie Chart. This is a very effective and efficient way of hooking up multiple event handlers in a single statement.

In JavaScript whenever you are attaching an event handler or callback to an event you're implementing the Observer pattern. JavaScript being an event-driven language, you simply cannot escape the Observer pattern. It is always present.

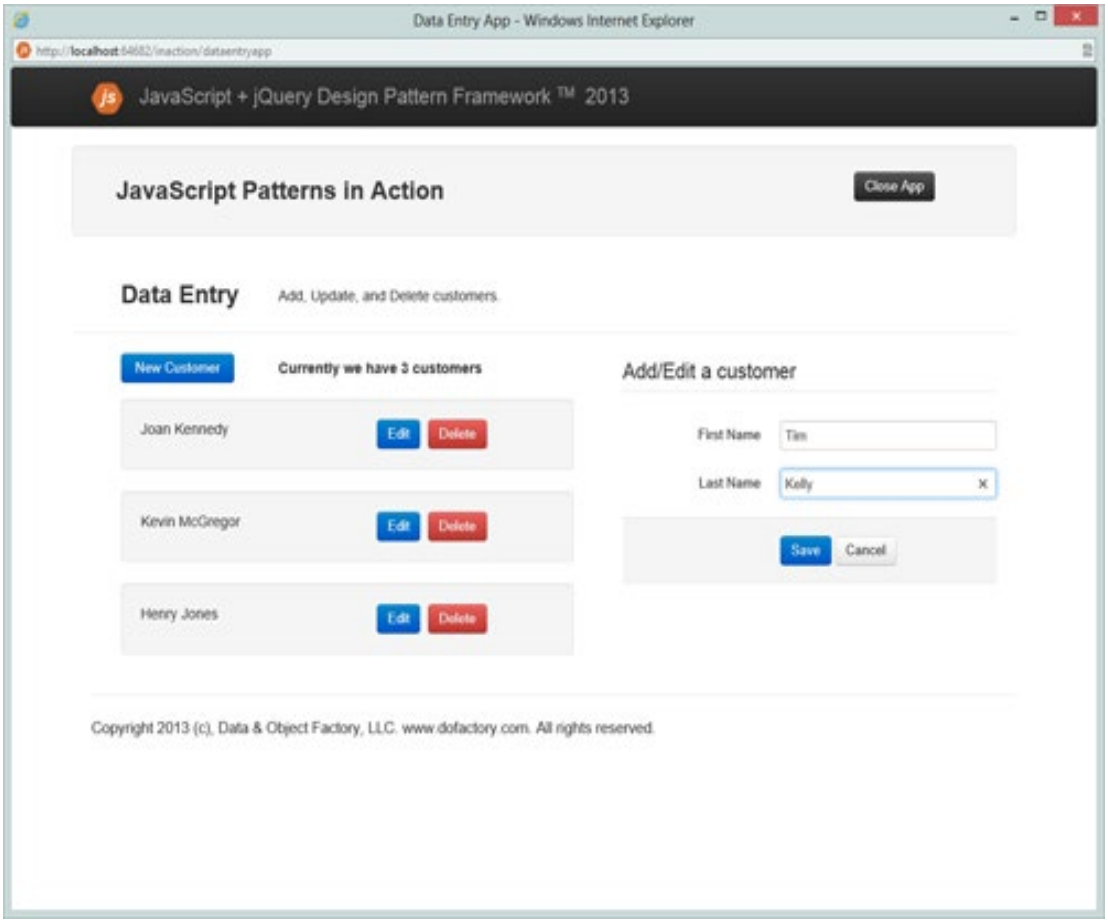
The real-time Chart is updated at a very high frequency: every 30 milliseconds. We could have written a tight loop without `setTimeout` but that would freeze the browser because it wouldn't have any time to catch up with other pending events in the event queue. The `setTimeout` call gives the browser breathing room and allows it to respond to all other events (including the repainting of the real-time chart). It is not quite 0 timeout (Zero Timeout pattern), but the idea and the effects are the same.

The Iterator pattern allows you to loop over an array or list of elements. The built-in `for` and `while` are iterators. Perhaps a more 'pure' Iterator implementation is the jQuery `each` method which is used in other Patterns in Action samples.

The script checks if the HTML5 *canvas* element is supported. It uses the Double !! idiom which ensures that `supportsCanvas` is a true Boolean.

Data Entry

Here is a screenshot of the app:



Data entry involves CRUD operations. CRUD stands for Create, Read, Update, and Delete which are the four basic operations that you will find in any app in which records are maintained (probably over 95% of the apps). These operations exactly map to the four database SQL statements: Insert, Select, Update, and Delete.

In this page we maintain customer records that have a first name and a last name. The page starts off with two existing records. You can add new records and edit or delete existing ones: fairly straightforward.

What is different is that everything takes place on the client for which we use the popular Backbone MVC framework. Backbone is built around the notion of REST services, but

since there is no server we built a Mock server using the open source fauxServer library. This is a valuable tool to help in automated testing. You'll see that doing CRUD on the client, as opposed to calling server pages, makes the app highly responsive.

The page elements are template driven. The templates are found in <script> tags with a `type="text/template"` attribute. The three templates are: `customers-template` which renders a list of customers, the `customer-template`, which renders a single record, and `form-template` which allows adding a new record or editing an existing record. Because of the `type="text/template"` attribute the browser will recognize these as templates rather than script. This is commonly used in Backbone templating. By the way, the SPA example (another Patterns-in-Action app) will use a different technique in which templates are loaded from disk.

Here are the three template scripts:

```
<script type="text/template" id="customers-template">
<div class="row">
    <div class="span2">
        <button class="btn btn-primary add"> New Customer </button>
    </div>
    <div class="span4" style="padding-top:5px;">
        <h4>Currently we have {{ count }} customers</h4>
    </div>
</div>
<div class="row">
    <div class="span6 offsethalf">
        <div class="customers"></div>
    </div>
</div>
</script>

<script type="text/template" id="customer-template">
    <div class="row">
        <div class="span3">{{ first }} {{ last }}</div>
        <div class="span2">
            <button class='btn btn-primary edit'>Edit</button>  
            <button class='btn btn-danger delete'>Delete</button>
        </div>
    </div>
</script>
```

```
<script type="text/template" id="form-template">
    <legend>Add/Edit a customer</legend>
    <fieldset>
    <div class='control-group'>
        <label class='control-label' for='first'>First Name</label>
        <div class='controls'>
            <input type='text' value="{{ first }}" class='input-large' id='first'>
        </div>
   </div>
   <div class='control-group'>
       <label class='control-label' for='last'>Last Name</label>
       <div class='controls'>
           <input type='text' value="{{ last }}" class='input-large' id='last'>
       </div>
   </div>
   <div class="form-actions">
       <button type="submit" class="btn btn-primary">Save</button>
       <button type="reset" class="btn">Cancel</button>
   </div>
   </fieldset>
</script>
```

Notice that the three script tags have a `type="text/template"` attribute.

By default Backbone uses the underscore.js templating engine. Its template token markers has a rather awkward `<% =  %>` syntax. We changed this to double braces `{{` and `}}` which is also more in line with Mustache, another popular templating engine. The `_.templateSettings` method is used to change this. You can see it in the code below.

Here is the code for the Data Entry app:

```
var Patterns = {
    // ** namespace pattern
    namespace: function (name) {
        // ** single var pattern
        var parts = name.split(".");
        var ns = this;

        // ** iterator pattern
        for (var i = 0, len = parts.length; i < len; i++) {
            // ** || idiom
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
```

```
        }

        return ns;
    }
};

// ** namespace pattern
// ** revealing module pattern
// ** singleton pattern
Patterns.namespace("InAction").DataEntry = (function () {

    // ** single var pattern
    // ** namespace pattern (Models, View, Routers)
    var Models = {};
    var Views = {};
    var Routers = {};

    var router;

    var start = function () {
        router = new Routers.Router();
        Backbone.history.start();
    }

    // Change template token markers to {{ and }}
    _.templateSettings = {
        interpolate: /\{\{(.+?)\}\}/g,
        evaluate: /\{\{(.+?)\}\}/g
    };

    // ** extend pattern
    // ** option hash idiom
    Routers.Router = Backbone.Router.extend({
        // ** init pattern
        initialize: function (options) {
            this.el1 = $("#customers-content");
            this.el2 = $("#form-content");
        },
        routes: {
            "": "main"
        },
        main: function () {
            this.el1.html(new Views.Customers().el);
        }
    });
```

```
// ** extend pattern
// ** option hash idiom
Models.Customer = Backbone.Model.extend({

    defaults: {
        first: "",
        last: ""
    }
});

// ** extend pattern
// ** option hash idiom
Models.Customers = Backbone.Collection.extend({
    model: Models.Customer,
    url: "customers"
});

// ** extend pattern
// ** option hash idiom
Views.Customers = Backbone.View.extend({
    template: _.template($('#customers-template').html()),

    events: {
        'click .add': 'add'
    },
    // ** init pattern
    initialize: function () {
        this.collection = new Models.Customers();
        // ** observer pattern
        this.collection.on('reset add update remove change', this.render, this);
        this.collection.fetch();
    },
    render: function (eventName) {

        // total customer count
        this.$el.html(this.template({ count: this.collection.length }));

        // ** iterator pattern
        this.collection.each(function (customer) {
            // ** option hash idiom
            var view = new Views.Customer({ collection: this.collection,
                                            model: customer });
            this.$el.append(view.render().el);
        }, this);

        return this;
```

```
        },
        add: function () {
            // ** option hash idiom
            var view = new Views.Customer.Form({ collection: this.collection,
                                                 model: new Models.Customer() });
            $("#form-content").html(view.render().el);
            $('#first').focus();
        }
    });

    // ** extend pattern
    // ** option hash idiom
    Views.Customer = Backbone.View.extend({
        className: 'well',
        // ** chaining pattern
        template: _.template($('#customer-template').html()),
        events: {
            'click .edit': 'edit',
            'click .delete': 'remove'
        },
        // ** init pattern
        initialize: function() {
            this.model.bind("destroy", this.close, this);
        },
        render: function () {
            this.$el.html(this.template(this.model.toJSON()));
            return this;
        },
        edit: function () {
            // ** option hash idiom
            var view = new Views.Customer.Form({ collection: this.collection,
                                                 model: this.model });
            $("#form-content").html(view.render().el);
            // ** chaining pattern
            // focus, but deselect the current value
            $('#first').focus().val('').val(this.model.get("first"));          },
        remove: function () {
            this.model.destroy();
        },
        close: function () {
            $(this.el).unbind();
            $(this.el).remove();
        }
    });

    // ** extend pattern
```

```
// ** option hash idiom
Views.Customer.Form = Backbone.View.extend({
    tagName: 'form',
    className: 'form-horizontal',
    // ** chaining pattern
    template: _.template($('#form-template').html()),
    events: {
        'submit': 'submit',
        'reset' : 'cancel'
    },
    render: function () {
        this.$el.html(this.template(this.model.toJSON()));
        return this;
    },
    submit: function (event) {
        event.preventDefault();

        if (this.model.isNew()) {     // add
            this.collection.create({
                first: this.$('#first').val(),
                last: this.$('#last').val()
            });

        } else {       // update
            var mod = this.collection.get(this.model.get("id"));

            this.model.set("first", this.$('#first').val());
            this.model.set("last", this.$('#last').val());
            var url = this.model.url;

            this.model.url = "customers";
            this.model.save();
            this.model.url = url;
        }
        this.$el.empty();
    },
    cancel: function (event) {
        event.preventDefault();
        this.$el.empty();
    }
});

// Helper: generate four random hex digits.
var S4 = function () {
    return (((1 + Math.random()) * 0x10000) | 0).toString(16).substring(1);
};
```

```
// Helper: generate a pseudo-GUID by concatenating random hexadecimal.
var guid = function () {
    return (S4() + S4() + "-" + S4() + "-" + S4() + "-" +
            S4() + "-" + S4() + S4() + S4());
};

// Used to mock database persistence
if (!window.databaseCustomers) {
    window.databaseCustomers = [{ id: "5f294421-08af-135f-1d22-583245fb67b5",
                                  first: "Joan", last: "Kennedy" },
                                { id: "461f92de-a7fc-a90d-4419-958423678d8f",
                                  first: "Kevin", last: "McGregor" }];
}

// server mock
// ** options hash idiom
fauxServer.addRoutes({
    listcustomers: {
        urlExp: "customers",
        httpMethod: "GET",
        handler: function (context) {
            context.data = databaseCustomers;
            return context.data;
        }
    },

    addCustomer: {
        urlExp: "customers",
        httpMethod: "POST",
        handler: function (context) {
            context.data.id = guid();
            databaseCustomers.push(context.data);
            return context.data;
        }
    },

    updateCustomer: {
        urlExp: "customers",
        httpMethod: "PUT",
        handler: function (context) {
            databaseCustomers.push(context.data);
            return context.data;
        }
    },
```

```
        deleteCustomer: {
            urlExp: "customers/:id",
            httpMethod: "DELETE",
            handler: function (context, bookId) {
                var len = databaseCustomers.length;
                for (var i = 0; i < len; i++) {
                    if (databaseCustomers[i].id === bookId) {
                        databaseCustomers.splice(i, 1);
                        break;
                    }
                }
            }
        }
    });

    return { start: start };
})();

$(function () {

    // ** facade pattern
    Patterns.InAction.DataEntry.start();
});
```

The patterns and idioms used in this code are:

- || and && idiom
- Option Hash idiom
- Namespace pattern
- Single var pattern
- Module pattern
- Extend pattern
- Init pattern
- Chaining pattern
- Iterator pattern
- Singleton pattern
- Observer pattern
- Façade pattern

Backbone is a powerful MVC framework. It is not very opiniated meaning there are many ways to solve the same problem. This is good and bad: good because it offers flexibility, and bad because it allows you to make bad design decisions. As a result, Backbone has a fairly steep learning curve. Unfortunately, the details of Backbone are beyond the scope of this program; our focus is on JavaScript and Patterns.

Most of the patterns listed above are used in a similar way as in the previous example (Dashboard) and will not be discussed here. The only differences are 1) the use of the Namespace pattern and 2) the Extend and Init patterns that are part of the Backbone framework.

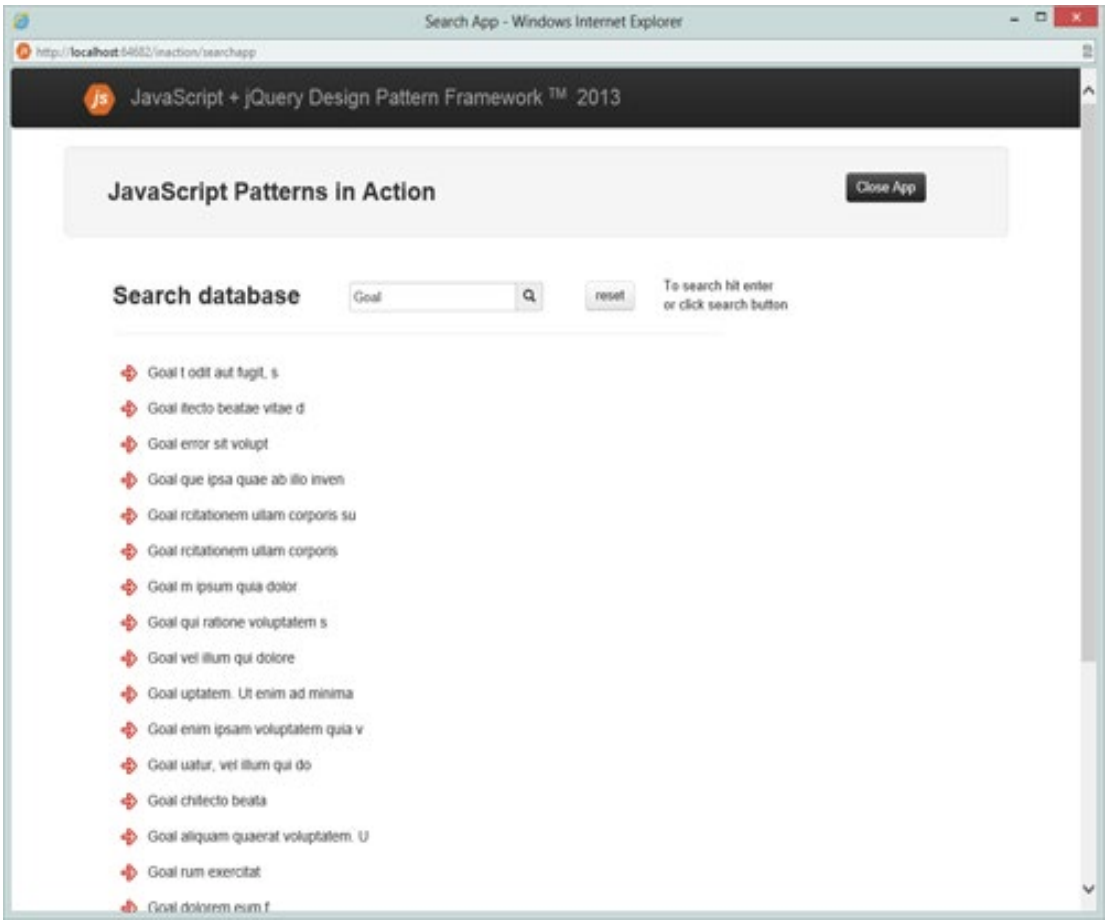
The Namespace pattern is used within the `DataEntry` module to organize the component categories in Backbone: `Models`, `Views`, and `Routers`. All model objects are stored in `Models`, all view objects in `Views`, and all router objects in `Routes`.

In Backbone there is usually a one-to-one relationship between models and views: for each model there is an associated view. The following naming convention has been adopted: when a model is named, for example, `Widget` we place it in `Models.Widget`. The associated view is also named `Widget` but we place it in `Views.Widget`. When we have collection of `Widget` model objects we pluralize it to `Widgets` and place this collection in `Models.Widgets`. The matching `View` will also be named `Widgets` and gets placed in `Views.Widgets`.

Backbone makes extensive use of the Extend pattern which lets you extend Backbone's base objects with your own custom objects. The `extend` method accepts a single argument which is an object literal with name/value pairs that are assigned as properties to the new object. This argument follows the Option Hash idiom. Backbone makes use of the Init Pattern which allows you to customize and complete the object initialization phase. The `initialize` method's argument is the same as the extend method so you have access to all object settings.

Search

Here is a screenshot of the app:



Search is important to most web apps. If you have a database with more than, say 100 records, you will want to consider some search, and possibly sort and filter, facility on your site. Since our app has no access to a real server, we will be using a Mock server that returns a random number of records generated with the help of a quote from Cicero, a Roman philosopher.

The page has two <script> templates. The `search-results-template` renders all search results and the `search-result-template` renders an individual record. Just as we did in the `DataEntry` example, the template token markers were changed from `<% = %>` to double braces `{{` and `}}` using a simple `_.templateSettings` call.

The Search App code is below:

```
var Patterns = {
    // ** namespace pattern
    namespace: function (name) {
```

```
        // ** single var pattern
        var parts = name.split(".");
        var ns = this;

        // ** iterator pattern
        for (var i = 0, len = parts.length; i < len; i++) {

            // ** || idiom
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

// ** namespace pattern
// ** revealing module pattern
// ** singleton pattern
Patterns.namespace("InAction").Search = (function () {

    // change template token markers to use double curly braces (just like Mustache):
    _.templateSettings = {
        interpolate: /\{\{(.+?)\}\}/g,
        evaluate: /\{\{(.+?)\}\}/g
    };

    // ** extend pattern
    // ** option hash idiom
    var Book = Backbone.Model.extend({
    });

    // ** extend pattern
    // ** option hash idiom
    var Books = Backbone.Collection.extend({
        model: Book,
        url: "search/:q"
    });

    // ** extend pattern
    // ** option hash idiom
    var Views = Backbone.View.extend({

        // ** chaining pattern
        template: _.template($('#search-results-template').html()),
        el: $("#searchresults"),
```

```
        // ** init pattern
        initialize: function (obj) {
            this.query = obj.query;

            this.books = new Books();
            // ** observer pattern
            this.books.on('reset', this.render, this); // will trigger render
            this.books.fetch({ data: { q: this.query } });
        },

        render: function (eventName) {
            this.$el.empty();
            // ** iterator pattern
            this.books.each(function (book) {
                this.$el.append(new View({ model: book }).render().el);
            }, this);

            return this;
        }
    });

    // ** extend pattern
    // ** option hash idiom
    var View = Backbone.View.extend({
        // ** chaining pattern
        template: _.template($('#search-result-template').html()),

        render: function (eventName) {
            this.$el.html(this.template(this.model.toJSON()));
            return this;
        }
    });

    // Random text helpers

    var randomInt = function (min, max) {
        return Math.floor(Math.random() * (max - min + 1)) + min;
    }

    // cicero: used in random string generator
    var cicero = "Sed ut perspiciatis unde omnis iste natus error sit voluptatem
accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo
inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim
ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia
consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro
```

```
quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed
quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat
voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis
suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum
iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur,
vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?";
    var ciceroLen = cicero.length;

    // server Mock
    fauxServer.addRoute("searchBook", "search/:q", "GET", function (context) {

        var q = context.data.q;
        var count = randomInt(6, 35);

        var array = [];

        // ** iterator pattern
        for (var i = 0; i < count; i++) {
            var from = randomInt(1, ciceroLen - 31);
            var to = from + randomInt(10, 30);
            // sub and trim
            var append = cicero.substring(from, to).replace(/^\s+|\s+$/g, '');
            array.push({ id: i, title: q + " " + append })
        }

        context.data = array;
        return context.data;
    });

    var start = function () {

        // focus on search text box and hook up enter keydown event
        // ** chaining pattern
        $("#search").focus().keydown(function (event) {
            if (event.keyCode == 13) {
                $("#searchbutton").click();
            }
        });

        // button click triggers search
        // ** observer pattern
        $("#searchbutton").on('click', function () {
            var q = $("#search").val();
            if (q) {
                new Views({ query: q });
            } else {
```

```
                alert("Please enter a search string");
                $("#search").focus();
            }
        });

        // ** observer pattern
        $("#reset").on('mousedown', function () {
            // ** chaining pattern
            $("#search").val("").focus();
            $("#searchresults").empty();
        });
    }

    return { start: start };
}());

$(function () {

    // ** facade pattern
    Patterns.InAction.Search.start();
});
```

The patterns and idioms used in this code are:

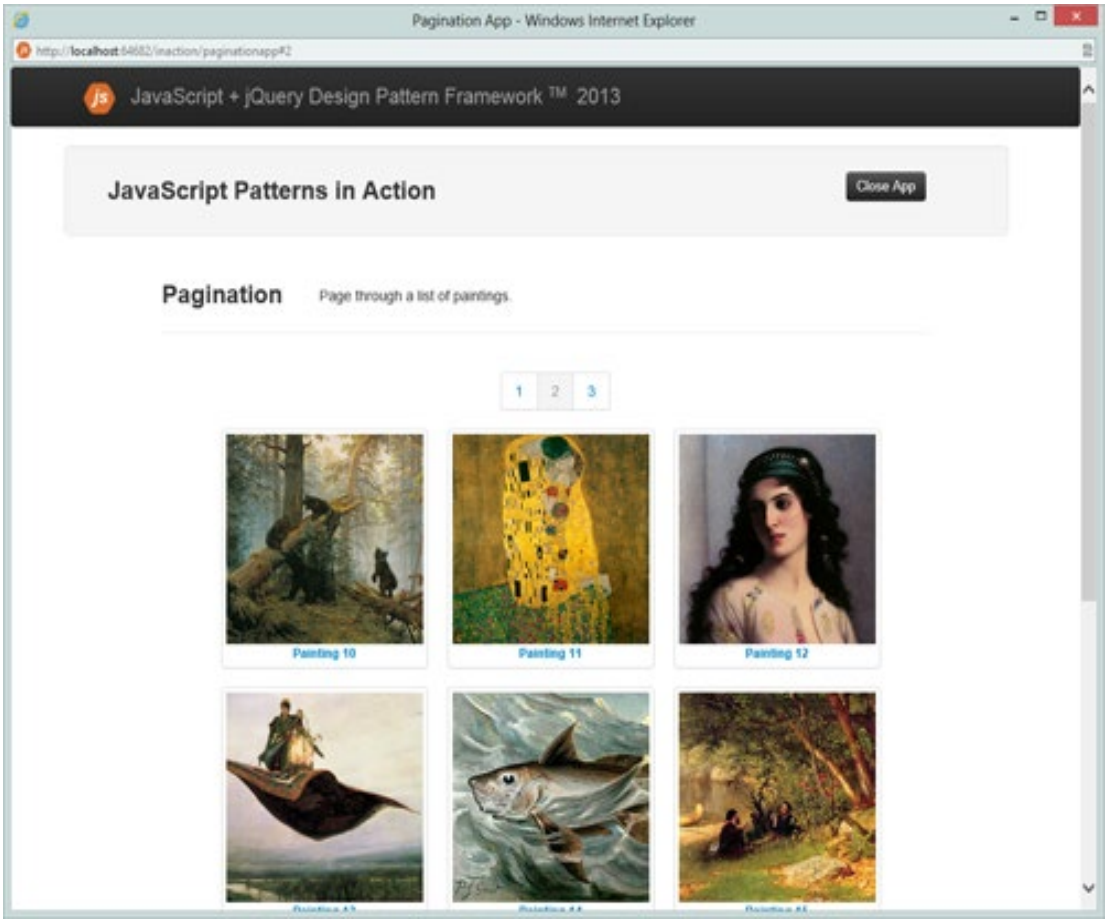
- || and && idiom
- Option Hash idiom
- Namespace pattern
- Single var pattern
- Module pattern
- Extend pattern
- Init pattern
- Chaining pattern
- Iterator pattern
- Singleton pattern
- Observer pattern
- Façade pattern

All of these patterns have already been described in the previous examples. This shows that the same patterns are commonly used over and over again. Once you develop the

skills to write patterns they will become second nature, almost as if they were native to the JavaScript language.

Pagination

Here is a screenshot of the app:



Most websites offer some form of pagination. There is a limit how many records can be displayed on a single page and breaking the record set up in pages makes a lot of sense. This pagination sample displays 9 well-known paintings per page. There are 3 pages total.

Like the previous examples this one uses Backbone, underscore templating (with changed template token markers), in-page templates (using <script> tags), and a Mock server. The two script templates are named `pictures-template` and `picture-template` respectively.

Here is the pagination code:

```
var Patterns = {
    // ** namespace pattern
    namespace: function (name) {
        // ** single var pattern
        var parts = name.split(".");
        var ns = this;

        // ** iterator pattern
        for (var i = 0, len = parts.length; i < len; i++) {
            // ** || idiomT
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

// ** namespace pattern
// ** revealing module pattern
// ** singleton pattern
Patterns.namespace("InAction").Pagination = (function () {

    // ** namespace pattern
    var Models = {};
    var Views = {};
    var Routers = {};

    // Change template token markers to {{ and }}
    _.templateSettings = {
        interpolate: /\{\{(.+?)\}\}/g,
        evaluate: /\{\{(.+?)\}\}/g
    };

    // ** extend pattern
    Routers.Router = Backbone.Router.extend({

        // ** init pattern
        initialize: function (options) {
            this.el = options.el;
        },
        routes: {
            "": "paginate",
```

```
            ":page": "paginate"
        },
        paginate: function (page) {

            // ** truthy/falsy idiom
            var p = page ? parseInt(page, 10) : 1;

            // ** chaining pattern
            this.el.html(new Views.Pagination({ page: p }).el);
            this.el.append(new Views.Pictures({ page: p }).el);
            this.el.append(new Views.Pagination({ page: p }).el);
        }
    });

    // ** extend pattern
    // ** option hash idiom
    Models.Picture = Backbone.Model.extend({
    });

    // ** extend pattern
    // ** option hash idiom
    Models.Pictures = Backbone.Collection.extend({
        model: Models.Picture,
        url: "pagination"
    });

    // ** extend pattern
    // ** option hash idiom
    Views.Pictures = Backbone.View.extend({

        tagName: "ul",
        className: "thumbnails",

        // ** init pattern
        initialize: function (options) {
            this.page = options.page;

            this.collection = new Models.Pictures();
            this.collection.bind("reset", this.render, this);
            this.collection.fetch({ data: { page: this.page } });
        },
        render: function () {

            // ** iterator pattern
            this.collection.each(function (picture) {
                var view = new Views.Picture({ model: picture });
```

```
            this.$el.append(view.render().el);
        }, this);

        return this;
    }
});

// ** extend pattern
// ** option hash idiom
Views.Picture = Backbone.View.extend({

    tagName: "li",
    template: _.template($('#picture-template').html()),

    render: function () {
        this.$el.html(this.template(this.model.toJSON()));
        return this;
    }
});

// ** extend pattern
// ** option hash idiom
Views.Pagination = Backbone.View.extend({

    tagName: "div",   // this is default

    className: "pagination pagination-centered",
    // ** init pattern
    initialize: function (options) {
        this.page = options.page;
        this.render();
    },

    render: function () {

        var pageCount = Math.ceil(pictures.length / 9);
        this.$el.css("padding-right", "100px");

        this.$el.append("<ul>");
        // ** iterator pattern
        for (var i = 0; i < pageCount; i++) {
            $('ul', this.el).append("<li " + ((i + 1) === this.options.page ?
                    " class='active'" : "") + "><a href='#" + (i + 1) + "'>" +
                    (i + 1) + "</a></li>");
        }
        this.$el.append("</ul>");
```

```
        return this;
    }
});

var pictures = [
    { id: 1, picture: "1a.jpg", title: "Painting 1" },
    { id: 2, picture: "2a.jpg", title: "Painting 2" },
    { id: 3, picture: "3a.jpg", title: "Painting 3" },
    { id: 4, picture: "4a.jpg", title: "Painting 4" },
    { id: 5, picture: "5a.jpg", title: "Painting 5" },
    { id: 6, picture: "6a.jpg", title: "Painting 6" },
    { id: 7, picture: "7a.jpg", title: "Painting 7" },
    { id: 8, picture: "8a.jpg", title: "Painting 8" },
    { id: 9, picture: "9a.jpg", title: "Painting 9" },
    { id: 10, picture: "10a.jpg", title: "Painting 10" },
    { id: 11, picture: "11a.jpg", title: "Painting 11" },
    { id: 12, picture: "12a.jpg", title: "Painting 12" },
    { id: 13, picture: "13a.jpg", title: "Painting 13" },
    { id: 14, picture: "14a.jpg", title: "Painting 14" },
    { id: 15, picture: "15a.jpg", title: "Painting 15" },
    { id: 16, picture: "16a.jpg", title: "Painting 16" },
    { id: 17, picture: "17a.jpg", title: "Painting 17" },
    { id: 18, picture: "18a.jpg", title: "Painting 18" },
    { id: 19, picture: "19a.jpg", title: "Painting 19" },
    { id: 20, picture: "20a.jpg", title: "Painting 20" },
    { id: 21, picture: "21a.jpg", title: "Painting 21" },
    { id: 22, picture: "22a.jpg", title: "Painting 22" },
    { id: 23, picture: "23a.jpg", title: "Painting 23" },
    { id: 24, picture: "24a.jpg", title: "Painting 24" },
    { id: 25, picture: "25a.jpg", title: "Painting 25" }
];

// server Mock
fauxServer.addRoute("picturesPage", "pagination", "GET", function (context) {

    // ** single var pattern
    var pageSize = 9;
    var start = (context.data.page - 1) * pageSize + 1;
    var finish = Math.min(start + pageSize, pictures.length + 1);

    context.data = _.filter(pictures, function (item) {
        var index = parseInt(item.id, 10);
        return (index >= start && index < finish);
    });
```

```
        return context.data;
    });

    // Startup code
    var start = function () {
        var router = new Routers.Router({ el: $("#pictures-content") });
        Backbone.history.start();
    }

    return {
        start: start
    };
})();


$(function () {

    // ** facade pattern
    Patterns.InAction.Pagination.start();
});
```

The idioms and patterns used in this example are:

- Truthy/falsy idiom
- || and && idiom
- Option Hash idiom
- Namespace pattern
- Single var pattern
- Module pattern
- Extend pattern
- Init pattern
- Chaining pattern
- Iterator pattern
- Singleton pattern
- Observer pattern
- Façade pattern

Each page makes a mocked Ajax server call and gets a number of pictures returned. Depending on how users use the app, you may consider caching the pictures already

returned to the client. This could be implemented using the Lazy Load pattern. To keep things simple this is not implemented in this example but wouldn't be hard to add.