Clojure data visualization

2012 July 20

Kevin J. Lynagh

Keming Labs

This handout accompanies *Declarative web data visualization using ClojureScript*, a talk at OSCON 2012 in Portland, Oregon. A recording of the talk, slides, and this handout are available on the Keming Labs website.

Slides: keminglabs.com/talks/
Email: kevin@keminglabs.com
Twitter: @lynaghk
Office: +1 888 502 1042

Clojure 101

Clojure is a dynamically-typed LISP dialect. Designed from the start as a hosted language, Clojure features easy interop with its host platform. Currently runtimes exist for the JVM, CLR, and JavaScript engines. Alpha-level community projects compile to (or otherwise run Clojure on) Lua, Python, and C.

This is just enough Clojure to understand the handout; for a more comprehensive overview, see Mark Volkmann's article: http://java.ociweb.com/mark/clojure/article.html.

Clojure has a relatively simple syntax compared to more common languages like Python or Java. Clojure programs consist of *forms*, which are parenthetical lists where the first item is something to be called and the remaining items are arguments. Compare these two function definitions and invocations; JavaScript on the left, Clojure on the right:

```
function hello(greet, x){
  console.log(greet + " " + x);
};
hello("heya", "reader");
```

```
(defn hello [greet x]
  (println (str greet " " x)))

(hello "heya" "reader")
```

Notice that the Clojure definition is a series of nested forms. The innermost form, `(str greet " " x)`, is the invocation of the `str` function with three arguments: the variable `greet`, the space string literal and the variable `x`. That form is nested in a call to `println`, which is itself the fourth item in the form that defines the hello function. Also notice that commas are whitespace in Clojure; you can use them to aid readability, but they are never required.

Clojure data literals

string	`"hello"`
keyword	`:key`
vector	`[1 2 3]`
map	`{:a 1 "b" 2}`
set	`#{1 2 3}`
list	`'(1 2 3)`
regex	`#"[a-z]+"`

In terms of semantics, the biggest difference between Clojure and more mainstream languages is Clojure's opionated approach toward state; its data structures (vectors, maps, and sets) are immutable. For instance, the `assoc` (associate) function doesn't add new key/value pairs to a map. Instead, it returns a new map that has all of the entries in the old map in addition to the new key/value pairs:

```
(let [m {:a 1 :b 2}]
  (assoc m :c 3) ;=> {:a 1, :b 2, :c 3}
  m)             ;=> {:a 1 :b 2}
```

Semicolons indicate comments in Clojure, and the `;;=>` is just an idiom depicting what the form on that line evaluates to.

Clojure offers a variety of reference types to model mutable state. An *atom*, for instance, references an immutable value and can be updated to later reference a different immutable value.

Atoms:

```
(let [a (atom 1)]
  @a  ;=> 1
  (swap! a + 5)
  @a) ;=> 6
```

The at sign is sugar for dereferencing: `@a` is really just a function call: `(deref a)`. Behind the scenes, the `swap!` call evaluates `(+ 1 5)`, which gives the new result.

C2 web visualization

C2 is a Clojure data visualization library inspired by the D3 JavaScript library. Both libraries construct data visualizations by mapping a collection of data to browser DOM elements (e.g., to make a bar graph you might map numbers to SVG `<rect>` elements with varying width attributes). Both C2 and D3 offer declarative APIs that avoid the tedious bookkeeping of looping and imperative DOM manipulation inherent in methods like `addChild`, `setAttribute`, or `style.setProperty`.

C2 differs from D3, in that you primarily build plain data structures *representing* the DOM rather than manipulating the DOM itself; compare D3 on the left with C2 on the right:

```
el.append("svg:g")
.attr("transform", "translate(0,20)")
.append("svg:circle")
.attr("cx", 5).attr("r", 10)
.style("background-color", "blue");
```

```
(dom/append! el
             [:g {:transform "translate(0, 20)"}
              [:circle {:cx 5 :r 10
                        :style {:background-color "blue"}}]])
```

The D3 code involves a lot of *code*: calling methods like `attr` and `style` that mutate live elements on the DOM. The C2 code is just one function, `append!`, which is passed the parent element and a vector representing the new child. This might seem like a minor distinction, but decoupling the *specification* of the desired markup from the *rendering* of that markup yields a great deal of flexibility.

Specifying the markup you want in terms of standard data structures (i.e., keywords, vectors, and maps) lets you leverage your programming language; you can use whatever functions you prefer to build up the markup, not just what is provided by library authors. Since you're working with plain data, you don't always need a live DOM; it is easy to test code without a browser or to render markup on the server side. Where you can use data, you can also use variables or nested forms, which allows you to inline appropriate concerns directly within markup. Consider this snippet from a todo list application:

Decoupling specification from rendering via intermediate data structures comes at a small performance hit, of course, but we've found tens of milliseconds a small price to pay for the greater flexibility and clearer code.

Full C2 TodoMVC implementation available at `https://github.com/lynaghk/c2-demos/tree/master/todoMVC`. Clojure's explicit mutation semantics enable simple one-way data → DOM binding (similar to Knockout.js); in this example whenever the `!todos` or `!filter` atoms update, this template automatically re-runs and updates the DOM.

```
(bind! "#main"
       [:section#main {:style (when (zero? (count @core/!todos))
                                {:display "none"})}
        [:input#toggle-all
         {:type "checkbox"
          :properties {:checked (every? :completed? @core/!todos)}}]
        [:label {:for "toggle-all"} "Mark all as complete"]
        [:ul#todo-list (unify (case @core/!filter
                                :active    (remove :completed? @core/!todos)
                                :completed (filter :completed? @core/!todos)
                                ;;default to showing all events
                                @core/!todos)
                              todo*)]])
```

Markup concerns are stated declaratively at the appropriate places. Note how the `#main` section is hidden when there are no todos and that the `#toggle-all` checkbox is checked only when everything in the todo list is completed.

The `unify` construct is a data structure that effectively means, "the children here should look like these data through this template function". In this example the data is a filtered list of todo items and the template is the `todo*` function (defined elsewhere). The rendering library will add, remove, and update nodes to satisfy the unify construct. This is the core DOM functionality of D3, but in C2 it is handled as a special case via a construct that can be nested in the context of surrounding markup.

This handout is a very brief and necessarily incomplete introduction to Clojure and web data visualization. I hope that it serves as a humble starting point for you to explore what I find to be very exciting topics. Good luck!

Warning: logic in templates is a hot-button issue for some folks. Use your judgment: factor complex logic into smaller functions, use explicit variables when it makes sense to name things, but also inline forms directly when a variable would be gratuitous.

Resources

D3: Awesome JavaScript data visualization library	`http://d3js.org`
C2: Clojure data visualization library	`https://github.com/lynaghk/c2`
Singult: data-driven DOM templating (usable from JS!)	`https://github.com/lynaghk/singult`
Mark Volkmann's Clojure intro	`http://java.ociweb.com/mark/clojure/article.html`
Himera: try ClojureScript directly in your browser	`http://himera.herokuapp.com/index.html`
The Joy of Clojure, by Houser and Fogus	`http://joyofclojure.com/`
Clojure/core's TV channel	`http://blip.tv/clojure`