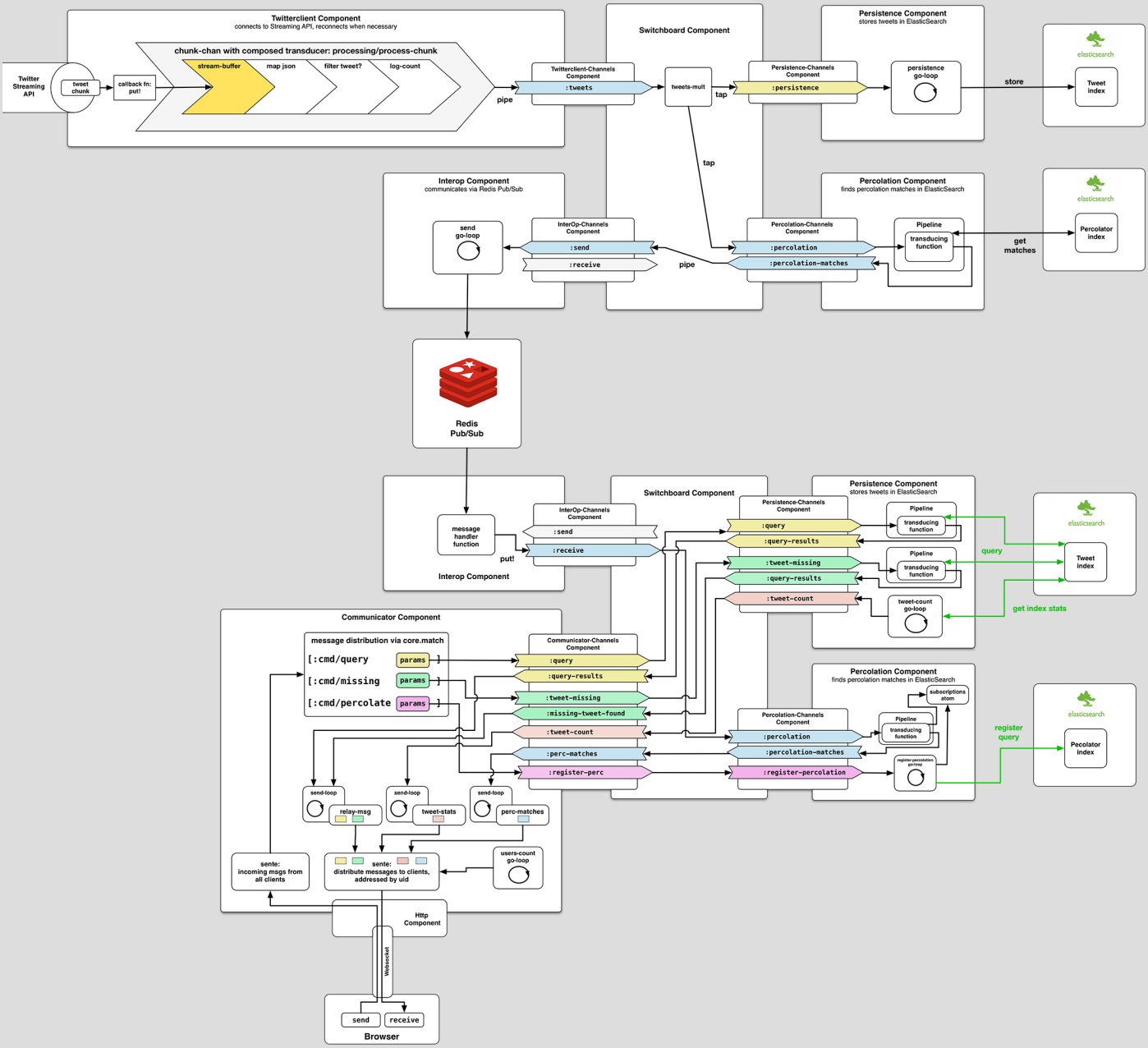
Building a System in Clojure and ClojureScript

Matthias Nehlsen



Building a System in Clojure (and ClojureScript)

Matthias Nehlsen

This book is for sale at http://leanpub.com/building-a-system-in-clojure

This version was published on 2016-07-25

Contents

1. Introduction

Hamburg, June 2016

A year and a half ago I started working on this book. The initial chapters just poured out of me, and then I realized that the kind of application that I was trying to build would greatly benefit from a library that makes composing a system out of individual components or subsystems that communicate via message passing much simpler. Then I had this gig that allowed me to explore the problem space further, and build a commercial application on top of it.

For a long time, I had wanted to get back to the book. But two things held me back:

1) I was waiting for inspiration to strike me once again, and then a few weeks later regain full conscience, noticing to my pleasant surprise that the book had completed itself in the meantime. Imagine my disappointment when I noticed that it doesn't work that way.

2) I was dreading having to write the documentation of the library without a mechanism in place that validates its usage, the messages passed around, and alterations to the managed component state. I had looked into **schema**[1] a few times, but somehow did not feel compelled to adopt it. No validation wasn't viable either, as then everything depends on the wording of the documentation, and that's a source of ambiguity that's best avoided.

Then, along came **clojure.spec**[2], and I was stunned how well it fit the bill. Within days, I had adapted the **systems-**[3] libraries and all of the sample applications, and I'm very excited how much more sense the entire approach makes after THE source of errors I had dealt with just fall by the wayside. The biggest problem had always been maps not structured as expected, and me as the developer having to keep those expectations in my head, rather than have the program do it for me. The capacity of my brain is way too small to keep such stuff in working memory.

In particular, I had those issues while working on my latest application, **iWasWhere**[4], which is an application for helping me get more done while being happier, in a geolocation-aware context. No worries, I'll get to that as it'll be one of the sample applications of this book. Among other things, **iWasWhere** is supposed to help me reach longer-term goals, such as finishing this very book, and I write it using the **systems-toolbox** libraries. Thus, I expect some cross-pollination between the book and the project. Anyway, building this application luckily put me back in the position of a user of the libraries, and that's a welcome change of perspective for moving it forward. And now with **clojure.spec**, this has become much more pleasant.

Now, this is going to be a **reboot** of the book project. For now, I will move all the existing chapters into the appendix, for reference. However, your time is probably better spent reading the new material, and then joining a discussion about it. I want this book to **help you** approach this beautiful language named Clojure, and I can do that best when you **let me know** what you found unclear, or where you would like additional explanations. To provide **feedback**, you can either contact me directly under matthias.nehlsen@gmail.com or create an issue in this book's **GitHub project**[5], ideally with file and line number.

[1] https://github.com/plumatic/schema

[2] http://clojure.org/about/spec

[3] https://github.com/matthiasn/systems-toolbox

[4] https://github.com/matthiasn/iWasWhere

[5] https://github.com/matthiasn/clojure-system-book

I imagine the **systems-toolbox** libraries to be helpful when you want to build applications like the ones described in this book. For anything that you find missing or where you want something improved, please also consider opening an issue on **GitHub**[6].

Now have fun playing around with the sample applications. I'm confident you will **learn the most** when checking out the code, changing stuff here and there, and build something different out of these applications. All of them here are compatible with **FigWheel**[7], by the way. It's instant feedback make coding all the more gratifying.

Okay, that's all, let me know how I can help!

Matthias

P.S. I'm curious about what you will build on top of these libraries. If you have a project you would like me to look at, shoot me an email. If it's an open source project, I'll be happy to do a code review for free. All else can be discussed.

[6] https://github.com/matthiasn/systems-toolbox

[7] https://github.com/bhauman/lein-figwheel

2. Example application: Counter

In **Clojure** (and **ClojureScript**), we like to use **persistent data structures**[1] because they are **immutable**[2]. Immutable data structures are great, because they make a program easier to reason about, and they make an entire class of potential bugs disappear: with immutable values, there is no possibility for **accidentally mutating** something. If you haven't seen Rich Hickey's talk "**The Value of Values**", you should watch it now, or read the **transcript**[3], and then continue with this chapter.

Okay, now you're familiar with Rich Hickey's thoughts on immutable data. Hopefully, we're now on the same page about their value. Ideally, when building a **UI**, we would want to create functions that take some **immutable** data and return some **HTML**. On the server side, that's exactly how you would have done it for years. With something like **Hiccup**[4], you can build such functions easily. However, this doesn't easily transfer to the **ClojureScript** world. Server-side rendering means having to reload the page when anything changes and long gone are the days in which that was sufficient. We can no longer expect that users will happily refresh a page and wait, wait, **wait**.

Rather, nowadays, we want to build highly **interactive** web applications that feel like **desktop applications** or **mobile apps**, rather than the typical thing of the early web, where you submit a form and wait for some page to appear, seconds later.

Ideally, we should be able to do the same thing on the client as we do on the server. Pass some data to a function, get some DOM subtree back, and move on. But for a long time there was no decent solution for this problem, which presumably has to do with the **DOM**[5] being this highly mutable construct, which UI frameworks such as **AngularJS 1.x**[6] use directly for attaching data and functionality. I tried to use AngularJS with **ClojureScript** a long time ago, and it just doesn't seem to work properly, in a way I'd call predictable.

Then, along came **React**[7], which changed everything. It allows us to write pure functions that we can feed immutable data, and that will build an entire DOM subtree out of the data every single time. Then, when a change in the data is detected, the render function is called again, generating the entire output. **React** will then do some diffing between the previous version and the latest version, in a virtual DOM, and deal with the messy DOM mutation to enact the detected changes in the "real" DOM. This approach may sound like a lot of work, but in reality, it's super fast, faster than anything we'd have to worry about in most cases.

Then there are ClojureScript libraries that make React available to us **Clojure(Script)** developers. The one that was available first was **Om**[8]. I prefer writing user interfaces in **Hiccup**, because I find this notation very terse and succinct, and it also makes for something that's

[1] https://en.wikipedia.org/wiki/Persistent_data_structure
[2] https://en.wikipedia.org/wiki/Immutable_object
[3] https://github.com/matthiasn/talk-transcripts/blob/4f17b730a370cf454266c90525ea5ff0d1f38098/Hickey_Rich/ValueOfValues.md
[4] https://github.com/weavejester/hiccup
[5] https://en.wikipedia.org/wiki/Document_Object_Model
[6] https://angularjs.org/
[7] https://facebook.github.io/react/
[8] https://github.com/omcljs/om

particularly easy to test, but more about that another time. Luckily, there's **Reagent**[9], which provides just that: a way to use **Hiccup** on the client, with React.

With Reagent, you can then start building applications, right from where its tutorials leave you. The promise there is that all you need to do is share a Reagent atom between the different parts of your application. Hmm, sounds simple enough, but every time I tried to build an evolving system around it, I ended up writing something that quickly became hard to maintain. Like, repeatedly. And pull-my-hair-out hard.

It turns out that those problems have nothing to do with Reagent. Rather, they were dealing with the same kinds of problems at **Facebook**, where React originated (via Instagram), so they came up with the **Flux pattern**[10].

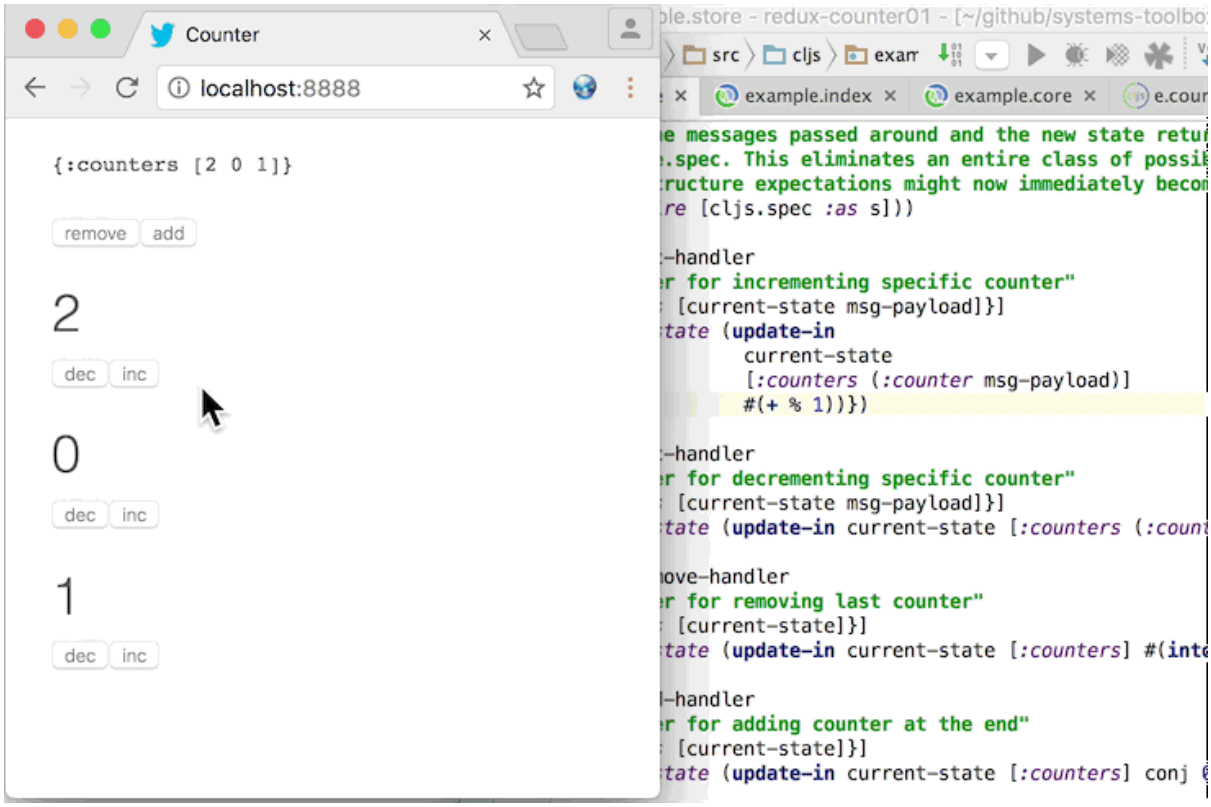
Flux deals with structuring the application in a way that all state mutation happens in a single place, rather than ad-hoc where UI functions have write access to data. The latter is just terribly hard to maintain and difficult to debug, and that's my experience when using Reagent also - which, after all, is only a thin wrapper on top of React.

Flux is an approach, not a library. However, there's **Redux**[11]. Redux is called a **predictable state container**. What's that? Well, basically a place where your data lives, and also the only place where that data changes. Every other part of the application only has read access. It's very helpful when you want to reason about an application. Without a structured approach to state management, I often find my mental complexity budget stretched beyond the point of breaking. As in, pulling my hair out because of not finding bugs that should not have existed in the first place.

As it turns out, the **systems-toolbox library**[12] allows for the same kind of approach suggested by Redux. There's a predictable state container, which is interacted with via **immutable messages** only, for example when clicking a button. Then, there are other components that observe the state in that container and get notified when it changes. These can for example then render an updated user interface. I found this to a be a helpful way for structuring applications, and I've written a handful in this pattern so far, probably most notably the latest incarnation of **BirdWatch**[13].

That application wouldn't make for a gentle introduction so let's instead start with something very simple. In the Redux tutorials, there's an **example with a counter**[14], where clicks on increment and decrement buttons change app state, which then again is re-rendered by React. Let's do the same thing in ClojureScript, using the **systems-toolbox** and **systems-toolbox-ui**[15] libraries.

[9] https://reagent-project.github.io/
[10] https://facebook.github.io/flux/
[11] https://github.com/reactjs/redux
[12] https://github.com/matthiasn/systems-toolbox
[13] https://github.com/matthiasn/BirdWatch
[14]
[15] https://github.com/matthiasn/systems-toolbox-ui



Counter Example

We start with three counters, which each can be incremented or decremented using a button, and we can also add or remove counters. Simple, right? You could certainly do the same with just Reagent and an atom, but in my experience, that doesn't scale when things get more complex than this. Let's instead keep state and UI separate and see what that looks like.

Let me briefly introduce the systems-toolbox model now. There's a **component**. A component is an entity that has a lifecycle. It has some state, and it reacts to messages. It then also has some **observable state**, which other parts of the application can look at, read-only.

In this example, there's the store component, let's just look at the code[16]. in the **example.store namespace**[17]:

```
(ns example.store
  "In this namespace, the app state is managed. One can only interact with the
   state by sending immutable messages. Each such message is then handled by a
   handler function. These handler functions here are pure functions, they
   receive message and previous state and return the new state.

   Both the messages passed around and the new state returned by the handlers
   are validated using clojure.spec. This eliminates an entire class of possible
   bugs, where failing to comply with data structure expectations might now
   immediately become obvious."
  (:require [cljs.spec :as s]))
```

[16]The links should always point to the latest version in the codebase and be in sync with the text in the book chapters. If you find that that is not the case, it means I messed up somewhere and need your help. Just send me an email to matthias.nehlsen@gmail.com and I will update the chapter text as quickly as possible. Thanks!

[17]https://github.com/matthiasn/systems-toolbox/blob/master/examples/redux-counter01/src/cljs/example/store.cljs

```
(defn inc-handler
  "Handler for incrementing specific counter"
  [{:keys [current-state msg-payload]}]
  {:new-state
   (update-in current-state [:counters (:counter msg-payload)] #(+ % 1))})

(defn dec-handler
  "Handler for decrementing specific counter"
  [{:keys [current-state msg-payload]}]
  {:new-state (update-in current-state [:counters (:counter msg-payload)] dec)})

(defn remove-handler
  "Handler for removing last counter"
  [{:keys [current-state]}]
  {:new-state (update-in current-state [:counters] #(into [] (butlast %)))})

(defn add-handler
  "Handler for adding counter at the end"
  [{:keys [current-state]}]
  {:new-state (update-in current-state [:counters] conj 0)})

(defn state-fn
  "Returns clean initial component state atom"
  [_put-fn]
  {:state (atom {:counters [2 0 1]})})

;; validate messages using clojure.spec
(s/def :redux-ex1/counter #(and (integer? %) (>= % 0)))
(s/def :cnt/inc (s/keys :req-un [:redux-ex1/counter]))
(s/def :cnt/dec (s/keys :req-un [:redux-ex1/counter]))

;; validate component state using clojure.spec
(s/def :redux-ex1/counters (s/coll-of integer? []))
(s/def :redux-ex1/store-spec (s/keys :req-un [:redux-ex1/counters]))

(defn cmp-map
  [cmp-id]
  {:cmp-id      cmp-id
   :state-fn    state-fn
   :state-spec  :redux-ex1/store-spec
   :handler-map {:cnt/inc    inc-handler
                 :cnt/dec    dec-handler
                 :cnt/remove remove-handler
                 :cnt/add    add-handler}})
```

Above, you can see that there are four handlers, for four different message types: `:cnt/inc`, `:cnt/dec`, `:cnt/add` and `:cnt/remove`.

Then, there's application state. The initial state is returned by the `state-fn`:

```
{:counters [2 0 1]}
```

Each of these three counters has an initial value, which can be changed by clicking the respective buttons. That's all there is to the app state. Each handler takes the `current-state` argument and returns the `:new-state` in the respective key in the returned **map**.

Then, as a recent addition to the library, there is also **validation** provided by the excellent **clojure.spec**[18], which for me changes everything in **Clojure** for the better. With it, we can specify precisely how both messages passed around and returned state changes are supposed to look like, and fail otherwise. This validation gives you the best of both worlds. You get the sanity check from a typed world, only better in some regards, and without all the clutter.

Next, let's have an eye on a UI component that makes use of this state to render something, and finally, look at how messages get passed back and forth between those components.

The UI functions are super simple. There are only three functions in the **example.counter-ui namespace**[19], `counter-view`, `counters-view`, and `cmp-map`:

```
(ns example.counter-ui
  (:require [matthiasn.systems-toolbox-ui.reagent :as r]
            [matthiasn.systems-toolbox-ui.helpers :as h]))

(defn counter-view
  "Renders individual counter view, with buttons for increasing or decreasing
   the value."
  [idx v put-fn]
  [:div
   [:h1 v]
   [:button {:on-click #(put-fn [:cnt/dec {:counter idx}])} "dec"]
   [:button {:on-click #(put-fn [:cnt/inc {:counter idx}])} "inc"]])

(defn counters-view
  "Renders counters view which observes the state held by the state component.
  Contains two buttons for adding or removing counters, plus a counter-view
  for every element in the observed state."
  [{:keys [current-state put-fn]}]
  (let [indexed (map-indexed vector (:counters current-state))]
    [:div.counters
     [h/pp-div current-state]
     [:button {:on-click #(put-fn [:cnt/remove])} "remove"]
     [:button {:on-click #(put-fn [:cnt/add])} "add"]
     (for [[idx v] indexed]
           ^{:key idx} [counter-view idx v put-fn])]))

(defn cmp-map
  [cmp-id]
  (r/cmp-map {:cmp-id  cmp-id
              :view-fn counters-view
              :dom-id  "counter"}))
```

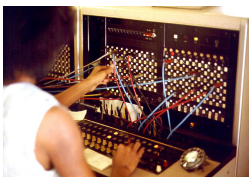
[18] https://clojure.org/about/spec

[19] https://github.com/matthiasn/systems-toolbox/blob/master/examples/redux-counter01/src/cljs/example/counter_ui.cljs

The `cmp-map` function returns a configuration map that systems-toolbox needs to start a component of this kind. In this case, that's a component that renders a small piece of UI into the element with the specified element ID in the DOM. There, it specifies that the counters-view function should be called to turn data into a piece of user interface.

This `counters-view` then gets passed the `current-state` and turns that into a tree structure of DOM elements, with add and remove buttons once, and then a `counter-view` for each indexed element in the counters in the `current-state`. Then, note that there's the `put-fn`, which we can call when the component is supposed to send something, so in this case when the respective button is clicked. Note that the index is used to identify which of the (initially three) counters to increment or decrement.

That's all there is to the UI component. Now let's look at how those components are wired together, in the `core` namespace. There's the `switchboard`, which you can think of like this:



Telephony switchboard

Someone connects a wire, and you can start talking. Only that here, the wires are **uni-directional**. Under the hood, there are **core.async**[20] channels connected to each other, but you don't need to worry about that for now.

Let's have a look at the **example.core namespace**[21]:

```
(ns example.core
  (:require [example.store :as store]
            [example.counter-ui :as cnt]
            [matthiasn.systems-toolbox.switchboard :as sb]))

(enable-console-print!)

(defonce switchboard (sb/component :client/switchboard))

(defn init
  []
  (sb/send-mult-cmd
    switchboard
    [[:cmd/init-comp (cnt/cmp-map :client/cnt-cmp)]
     [:cmd/init-comp (store/cmp-map :client/store-cmp)]
     [:cmd/route {:from :client/cnt-cmp :to :client/store-cmp}]
     [:cmd/observe-state {:from :client/store-cmp :to :client/cnt-cmp}]]))

(init)
```

First, the **switchboard** is created. Then, we send a message to the switchboard, with a vector containing multiple commands. We start with initializing the `:client/cnt-cmp`

[20] https://github.com/clojure/core.async

[21] https://github.com/matthiasn/systems-toolbox/blob/master/examples/redux-counter01/src/cljs/example/core.cljs

and `:client/store-cmp` components, which are responsible for UI and state management, respectively. The order here is not relevant, as these components don't need to know about each other anyway.

Then, we **route** messages from the UI component to the store component by using `:cmd/route`. Routing means that a connection is made for all messages for which there is a handler, so here `:cnt/inc`, `:cnt/dec`, `:cnt/add` and `:cnt/remove`, as we've seen in the `:handler-map` earlier. With this, whenever we use the `put-fn` inside the UI and send a message of any of these types, the store will receive it.

So far so good. Next, we need the UI to observe the state of the store component, which happens when sending the `:cmd/observe-state` message. Whenever the state of the`:client/store-cmp` changes, the UI will now have a copy of the change in it's `local` atom.

That's all there is to it. Now, this example has been quite simple. However, you can build much more complex applications in the same style. Very recently, this approach has become much more viable thanks to **clojure.spec**[22], which is a great addition to my development toolbox. You should use it in your projects, too. If you have not heard the latest **Cognicast**[23] where Rich Hickey talks about it, you should do that right now.

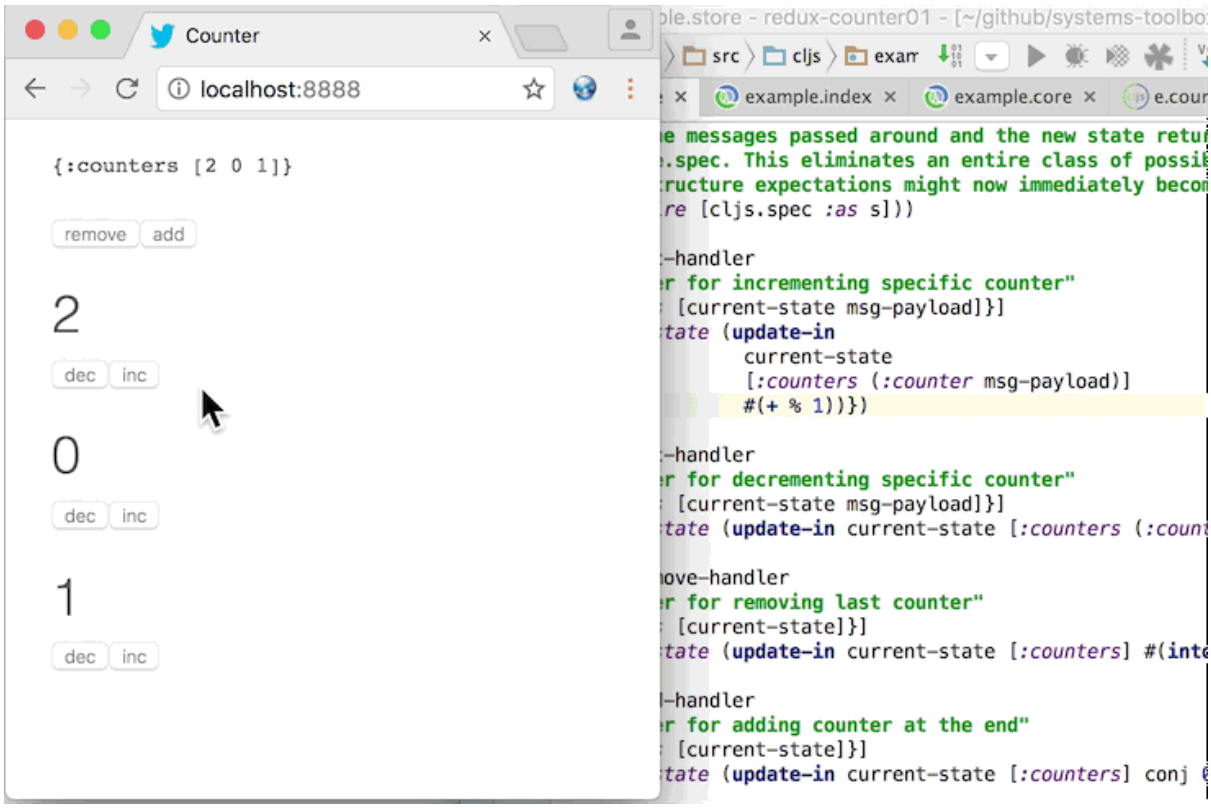
Note that not only does **clojure.spec** allow us to validate our app-specific data structures - it is also used dynamically in the switchboard when wiring components, so that validation takes application state into account. This dynamic validation is powerful, and would be difficult to achieve with a type system. Whenever there's another `cmp-id` that the switchboard has initialized, the set of possible values is updated, so that once it comes to `route` and `observe-state`, only valid component ids can be used. Try changing a component ID and you'll see an error message that is surprisingly not terrible for Clojure. Yeah, I don't like typical error messages in Clojure, and anything that makes the situation better is much appreciated.

Oh, I should also note **Figwheel**[24]. Applications built with the **systems-toolbox** are compatible with figwheel, with a page reload on every code change, while preserving application state. The reload mechanism is very useful during development, especially when you have some login. It's tedious without, where you have to recreate the app state after reloading the page so that you can judge the little change you made. Not so here, you make the change, and the page reloads automatically, while retaining the application state.

[22] https://clojure.org/about/spec

[23] http://blog.cognitect.com/cognicast/

[24] https://github.com/bhauman/lein-figwheel



Figwheel in Action

Also, this is incredibly useful when doing CSS changes. Usually, you'd probably do tiny changes in the developer tools until you have achieved the desired effect. But with Figwheel, the page will also reload while retaining app state, typically without any jumpiness.

Have a look and try it out for yourself. For that, I'd like you to clone the repository and run the application as follows:

```
lein run
```

And in an additional terminal:

```
lein figwheel
```

And now go to the store and change what happens when clicking the `inc` button. Where before, the value would be incremented by one, we could now have it increment by 11, like this:

```
(defn inc-handler
  "Handler for incrementing specific counter"
  [{:keys [current-state msg-payload]}]
  {:new-state (update-in current-state [:counters (:counter msg-payload)] #(+ % 11))})
```

After saving `store.cljs`, you'll briefly notice the figwheel logo overlayed on top of the page, and next, you click the button and increment the previous counter value, only that now you'll add 11 or whatever else you chose as the number there in your changed `inc-handler` function.

You can probably imagine how useful that can be when you build anything more complex. And over the next couple of chapters, I will show you different examples of more complex applications using the same pattern, only then composing more complex behavior out of the same predictable handler functions. By the way, these handler functions are easily testable because

they are pure, acting on immutable data and returning new values, rather than mutating some existing state. We'll get to that in a later chapter.

Now check out the example application, play around with it, and let me know what you think. The **systems-toolbox**[25] has helped me build these applications so far:

- **BirdWatch**[26]
- **iWasWhere**[27]
- a client project
- **trailing mousepointer example**[28]
- **redux counter example**[29]

It may help you build your application, too.

P.S. I needed some integration test for the **systems-toolbox-ui**[30] library, something running in an actual browser. So I wrote some tests running the example discussed above, clicking the buttons, and then asserting that they change as expected. You can run those tests and see for yourself, the instructions are **here**[31].

[25] https://github.com/matthiasn/systems-toolbox
[26] https://github.com/matthiasn/BirdWatch
[27] https://github.com/matthiasn/iWasWhere
[28] https://github.com/matthiasn/systems-toolbox/tree/master/examples/trailing-mouse-pointer
[29]
[30] https://github.com/matthiasn/systems-toolbox-ui
[31] https://github.com/matthiasn/systems-toolbox-ui

3. WebSocket Latency Visualization Example

Communication between backend and web applications via **WebSockets**[1] is an integral part of delivering a rich user experience. With that addition, it is much easier to push new information to the user at any given time, without having to resort to constant polling.

But how fast IS that communication? Let's find out. The next sample application for the **systems-toolbox**[2] deals with just that, visualizing the latency for messages sent from client to server and back.

[1] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[2] https://github.com/matthiasn/systems-toolbox

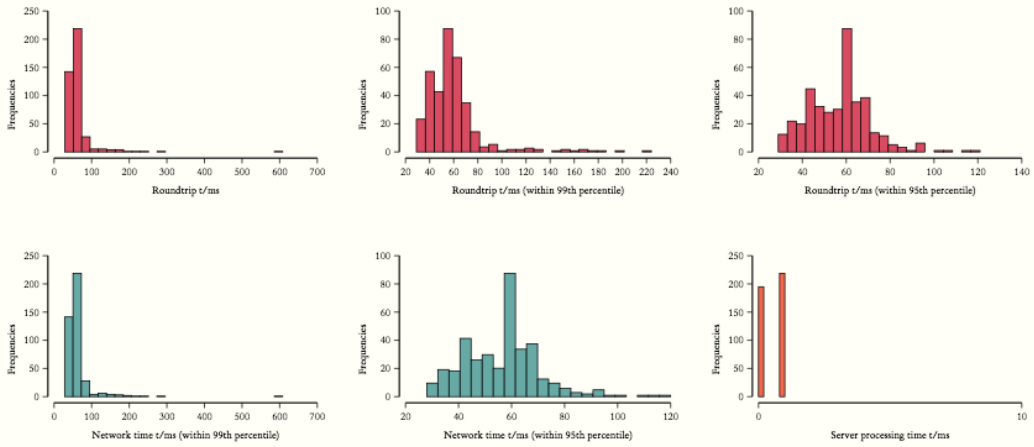
WebSockets Latency Visualization

Matthias Nehlsen

WebSockets bring bi-directional communication to the browser. This enables you to deliver interactive, real time web applications where all the data is as of right now, rather than always being outdated, and then constantly refreshed.

But how fast is this transport mechanism? Let's have a look. You may have noticed the circle around the mouse pointer on this page, or in fact the two circles, where one of them appears to follow the other. Both represent your last mouse position, only that one was sent to and returned from the server in the meantime. This gives you an intuition for how long it takes. Also, with your movement of the mouse, you generate data for the histograms below, which show the roundtrip duration:

What happens here is that movements of the mouse (or your finger on your mobile device) are captured. The more reddish one is then painted immediately, whereas the bluish one is painted after the event is sent to a server somewhere in Germany, and then back to wherever you are.



Screenshot

Check out the live demo **here**[3]; it'll give you some additional information about the application. There was a previous version of this example, but with **clojure.spec**[4] released, it was a good time to revisit this application and have it fully validated.

By the way, **clojure.spec**[5] came at a **crucial time** for me. This kind of validation was missing in the systems-toolbox (and more broadly in Clojure, too), and that made me question the whole approach, especially after fighting with annoying bugs in my latest application, **iWasWhere**[6] (which I'll introduce in a subsequent chapter). But now with clojure.spec, the entire class of those annoying bugs is gone for good. In a matter of a little over a week, I upgraded all my applications plus the systems-toolbox libraries to use clojure.spec, and I'm now more convinced about the approach than ever. You **can** build applications this way, and **stay sane** at the same time.

We'll look into validation in this chapter, too. But let's get started with the application itself. Here, we have a couple of different components:

On the client:

- there's the `:client/mouse-cmp` component that shows the position of the mouse, both locally and for the message coming back from the server
- there's the `:client/store-cmp`, which holds the client-side state
- there's the `:client/histogram-cmp` UI component for visualizing the round trip times as histograms
- there's `:client/info-cmp` UI component that shows some information about the app
- also, there are components for visualizing message flow, and for showing some JVM stats: `:client/observer-cmp` and `:client/jvmstats-cmp`

On the server:

- there's the `:server/ptr-cmp` component, which keeps a counter of all messages passed through since application startup, and returns each mouse position message to the client where it originated, plus, upon request, a history of mouse positions from all connected clients
- then, there's also the `:server/metrics-cmp` component for gathering some stats about the JVM, which get broadcast to all connected clients

On both sides, there are **Sente**[7] components for establishing **bi-directional communication** between client and server. These ready-to-use components are provided by the **systems-toolbox-sente**[8] library, and you can use them in your projects, too, with a simple import and no more than a handful of lines of code.

The store component on the client, which holds the client-side state, is then observed by the histogram, the mouse moves, and the info components; these three render something based on what's in the state that they observe.

[3] http://systems-toolbox.matthiasnehlsen.com/
[4] https://clojure.org/about/spec
[5] https://clojure.org/about/spec
[6] https://github.com/matthiasn/iWasWhere
[7] https://github.com/ptaoussanis/sente
[8] https://github.com/matthiasn/systems-toolbox-sente

The communication between these components is comparable to what was introduced in the previous chapter. What's new here is the `sente-cmp`. Let's have a brief look what **WebSockets**[9] are. They give us a way to establish a very low latency **bi-directional** connection between client and server. It's not HTTP but instead its own protocol. WebSockets are well supported from IE 10 on, and in all other recent browsers. Some critics say that they may be problematic because firewalls and reverse proxies might have to be reconfigured. Well, that might indeed problematic if (and only if) your OPS people are incompetent. But most likely they are not, so it's only a matter of communication (and some upfront planning) to get this potential hurdle out of the way.

Other than that potential issue with your firewall, there appears to be no downside to using **WebSockets**, and plenty of upsides. You absolutely need to be able to send messages from server to client at any time if you want to build a modern, responsive UI. Sure, you could also use **Server-sent Events (SSE)**[10] for the server -> client direction, and send messages from client to server the way you'd normally do: via REST calls, for each message. That may work; it may also be too expensive if you want to send messages often. For the use case in this very example, with the mouse positions, the user experience would likely be quite poor.

WebSockets are also nice because you get an ordering guarantee, which would be much harder with REST calls. Another aspect not to underestimate is that with REST calls, you need to think about authentication on every single request, where you do it once for a WebSockets connection.

3.1 :client/mouse-cmp

Anyway, let's look at some code, starting with where the messages originate in our example, the `:client/mouse-cmp` component, and its respective **namespace**[11]:

```
(ns example.ui-mouse-moves
  (:require [matthiasn.systems-toolbox-ui.reagent :as r]
            [matthiasn.systems-toolbox-ui.helpers :refer [by-id]]))

;; some SVG defaults
(def circle-defaults {:fill "rgba(255,0,0,0.1)
" :stroke "rgba(0,0,0,0.5)"
                      :stroke-width 2 :r 15})
(def text-default {:stroke "none" :fill "black" :style {:font-size 12}})
(def text-bold (merge text-default {:style {:font-weight :bold :font-size 12}}))

(defn mouse-hist-view
  "Render SVG group with filled circles from a vector of mouse positions in state."
  [state state-key stroke fill]
  (let [positions (map-indexed vector (state-key state))]
    (when (seq positions)
      [:g {:opacity 0.5}
       (for [[idx pos] positions]
         ^{:key (str "circle" state-key idx)}
```

[9] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[10] https://en.wikipedia.org/wiki/Server-sent_events

[11] https://github.com/matthiasn/systems-toolbox/blob/master/examples/trailing-mouse-pointer/src/cljs/example/ui_mouse_moves.cljs

```
         [:circle {:stroke       stroke
                   :stroke-width 2
                   :r            15
                   :cx           (:x pos)
                   :cy           (:y pos)
                   :fill         fill}])])))

(defn trailing-circles
  "Displays two transparent circles. The position of the circles comes from
   the most recent messages, one sent locally and the other with a roundtrip to
   the server in between. This makes it easier to visually detect any delays."
  [state]
  (let [local-pos (:local state)
        from-server (:from-server state)]
    [:g
     [:circle (merge circle-defaults {:cx (:x local-pos)
                                      :cy (:y local-pos)})]
     [:circle (merge circle-defaults {:cx (:x from-server)
                                      :cy (:y from-server)
                                      :fill "rgba(0,0,255,0.1)"})]]))

(defn mouse-view
  "Renders SVG with both local mouse position and the last one returned from the
   server, in an area that covers the entire visible page."
  [{:keys [observed local]}]
  (let [state-snapshot @observed
        mouse-div (by-id "mouse")
        update-dim
        #(do (swap! local assoc :width (- (.-offsetWidth mouse-div) 2))
             (swap! local assoc :height (aget js/document "body" "scrollHeight")))]
    (update-dim)
    (aset js/window "onresize" update-dim)
    [:div
     [:svg {:width  (:width @local)
            :height (:height @local)}
      (trailing-circles state-snapshot)
      (when (-> state-snapshot :show-all :local)
        [mouse-hist-view state-snapshot :local-hist
         "rgba(0,0,0,0.06)" "rgba(0,255,0,0.05)"])
      (when (-> state-snapshot :show-all :server)
        [mouse-hist-view state-snapshot :server-hist
         "rgba(0,0,0,0.06)" "rgba(0,0,128,0.05)"])]]))

(defn init-fn
  "Listen to onmousemove events for entire page, emit message when fired.
  These events are then sent to the server for measuring the round-trip time,
  and also recorded in the local application state for showing the local mouse
  position."
  [{:keys [put-fn]}]
```

```
  (aset js/window "onmousemove"
        #(put-fn [:mouse/pos {:x (.-pageX %) :y (.-pageY %)}]))
  (aset js/window "ontouchmove"
        (fn [ev]
          (let [t (aget (.-targetTouches ev) 0)]
            (put-fn [:mouse/pos {:x (.-pageX t) :y (.-pageY t)}])
            #_(.preventDefault ev)))))

(defn cmp-map
  "Configuration map for systems-toolbox-ui component."
  [cmp-id]
  (r/cmp-map {:cmp-id  cmp-id
              :view-fn mouse-view
              :dom-id  "mouse"
              :init-fn init-fn
              :cfg     {:msgs-on-firehose true}}))
```

Here, we have a UI component that covers the entire page. This is facilitated by the following CSS:

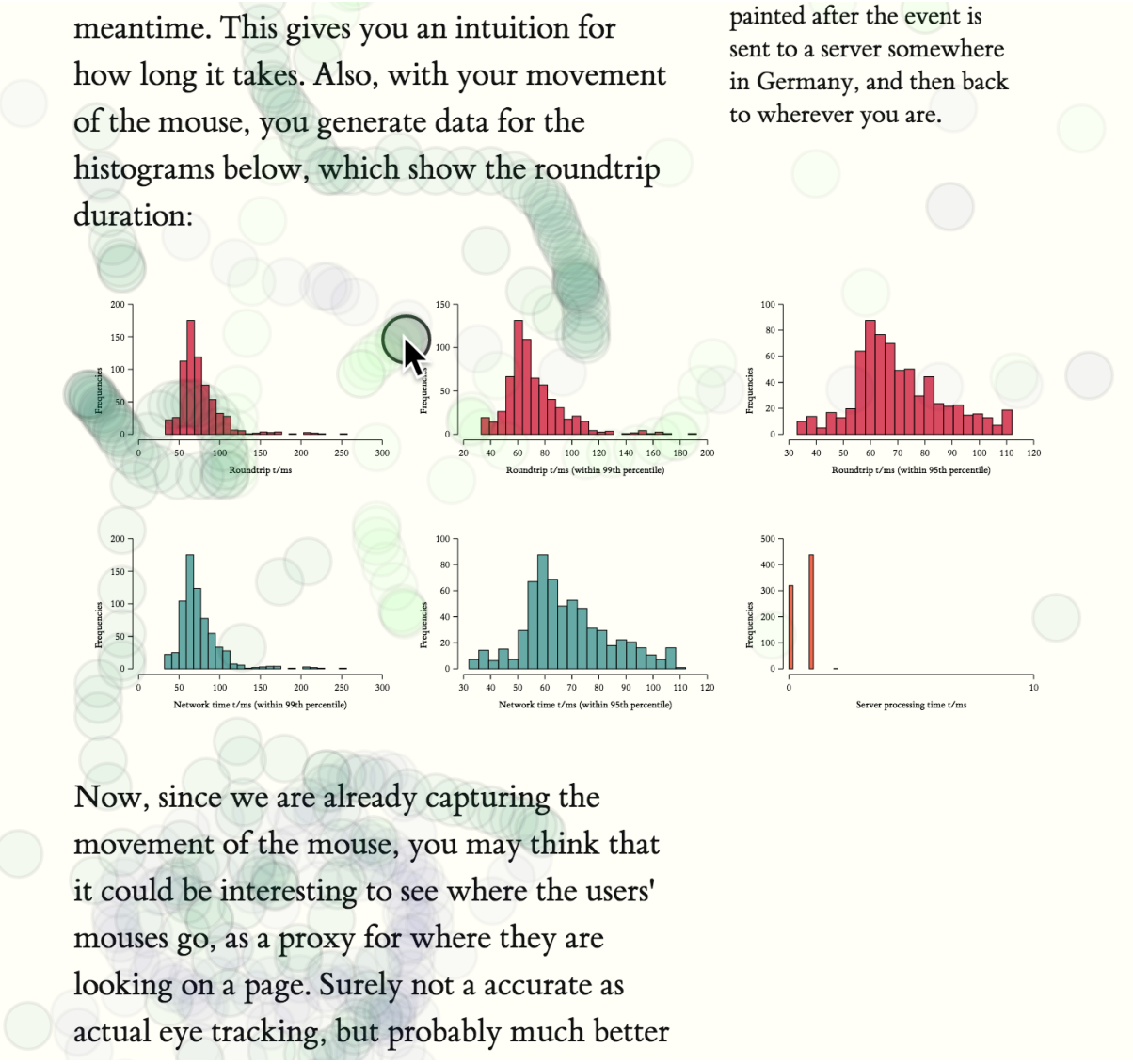
```
#mouse {
    position: absolute;
    top: 0;
    width: 100%;
    pointer-events: none;
    z-index: 10;
    margin-left: -12.5%;
}
```

Note that we want this transparent element on top, covering the rest of the page, which is what the `z-index` does. Also, we want `pointer-events` to reach the elements below, for example for clicking links or buttons, so we set them to `none`.

Then, in the `init-fn`, we set `ontouchmove` and `ontouchmove` event handlers, which get called when these events are fired anywhere on the page. We could also more specifically handle these events in the component's div, but then the `pointer-events` would not be available for elements below the `mouse-view` element, such as for clicking a button. Then, whenever an event is fired, a messaged is sent with the mouse position. This message will be received by the client side store directly, and also via the server side, where it'll be enriched with some additional data.

Then, the rendering of the **SVG**[12] covering the entire page is done in the `mouse-view` function, which adapts the size of the element when `onresize` element fires. Here, the `trailing-circles` function is called, which renders the two circles. This SVG rendering is trivial to achieve with Reagent. You can see that we just create a group with two circles, each with a distinct position based on the last known message. Fast movements will then reveal latency, as you'll see how the messages coming back from the server are lagging behind. Then, there are two calls to the `mouse-hist-view` function, which renders either a local history or the last moves of all clients, as you hopefully have seen when playing around with the live demo of the application. If not, here's what that looks like:

[12] https://www.w3.org/Graphics/SVG/



Screenshot

In the screenshot above, you can see green circles for the mouse moves captured locally, and charcoal ones for those from all clients.

Let's go through the **namespace**[13], function by function, starting from the bottom:

```
(defn cmp-map
  "Configuration map for systems-toolbox-ui component."
  [cmp-id]
  (r/cmp-map {:cmp-id  cmp-id
              :view-fn mouse-view
              :dom-id  "mouse"
              :init-fn init-fn
              :cfg     {:msgs-on-firehose true}}))
```

The `cmp-map` function creates the component map, which is like a blueprint that tells the switchboard how to fire up the component. The **UI** part is done by calling `r/cmp-map`, which

[13] https://github.com/matthiasn/systems-toolbox/blob/master/examples/trailing-mouse-pointer/src/cljs/example/ui_mouse_moves.cljs

is the main function in the **systems-toolbox-ui** library. Once the returned map is sent to the switchboard, a component will be initialized that renders the `mouse-view` function into the **DOM element** with the `"mouse"` ID.

Then, there's the `init-fn`:

```
(defn init-fn
  "Listen to onmousemove events for entire page, emit message when fired.
  These events are then sent to the server for measuring the round-trip time,
  and also recorded in the local application state for showing the local mouse
  position."
  [{:keys [put-fn]}]
  (aset js/window "onmousemove"
        #(put-fn [:mouse/pos {:x (.-pageX %) :y (.-pageY %)}]))
  (aset js/window "ontouchmove"
        (fn [ev]
          (let [t (aget (.-targetTouches ev) 0)]
            (put-fn [:mouse/pos {:x (.-pageX t) :y (.-pageY t)}])
            #_(.preventDefault ev)))))
```

This function takes care of registering handler functions for all mouse movements (and also touch movement, for that matter) for the entire window. By doing that here, for the entire window, we can get away with the `mouse-view` element not getting any mouse movement events, which is required for still reacting to clicks in elements that are in fact covered by it, since it spans the entire page. When such an event is encountered, a `:mouse/pos` message is sent, which then happens to be received by both the `:client/store-cmp` and the `:server/pos-cmp`. Not that this component needs to be concerned with that in any way, though - there's proper decoupling between them.

You can see how those messages are supposed to look like in the respective **specs**:

```
(s/def :ex/x pos-int?)
(s/def :ex/y pos-int?)

(s/def :mouse/pos
  (s/keys :req-un [:ex/x :ex/y]))
```

If you still haven't heard Rich Hickey talk about **clojure.spec**[14] on the **Cognicast**[15], you seriously need to do that now. **clojure.spec** has many useful properties. Among them is that you'll immediately know if you've broken your application with some recent change, as the system would throw an error immediately, rather than drag that problem along and blow up in your face somewhere else, where you'll have a hard time figuring out where it originated. What's also very useful is that when you come back to some code you wrote some time ago and wanted to know what a message is supposed to look like, you don't have to print it out and infer what the rules may be. No, instead you just look at the piece of code that's run when validating the message, it'll tell you all nitty-gritty details of what the expectations are. Much nicer.

Next, let's have a look at the `mouse-view` function, which is responsible for rendering the UI component:

[14] http://clojure.org/about/spec

[15] http://blog.cognitect.com/cognicast/103

```
(defn mouse-view
  "Renders SVG with both local mouse position and the last one returned from the
   server, in an area that covers the entire visible page."
  [{:keys [observed local]}]
  (let [state-snapshot @observed
        mouse-div (by-id "mouse")
        update-dim
        #(do (swap! local assoc :width (- (.-offsetWidth mouse-div) 2))
             (swap! local assoc :height (aget js/document "body" "scrollHeight")))]
    (update-dim)
    (aset js/window "onresize" update-dim)
    [:div
     [:svg {:width  (:width @local)
            :height (:height @local)}
      (trailing-circles state-snapshot)
      (when (-> state-snapshot :show-all :local)
        [mouse-hist-view state-snapshot :local-hist
         "rgba(0,0,0,0.06)" "rgba(0,255,0,0.05)"])
      (when (-> state-snapshot :show-all :server)
        [mouse-hist-view state-snapshot :server-hist
         "rgba(0,0,0,0.06)" "rgba(0,0,128,0.05)"])]]))
```

Note that this component gets passed a map with the `observed` and `local` keys. The `observed` key is an atom which holds the state of the component it observes. Here, this is always the latest snapshot of the `store-cmp`. The `local` atom contains some local state, such as the width of the SVG for resizing. Note that we're detecting the width on every call to the function, and also in the `onresize` callback of `js/window`. This ensures that the mouse div fills the entire page, while working with the correct pixel coordinate system. One could instead also use a viewBox, like this: `{:width "100%" :viewBox "0 0 1000 1000"}`. However, that would not work correctly in this case as the mouse position would not be aligned with the circles here.

Next, we have the `trailing-circles` function:

```
(defn trailing-circles
  "Displays two transparent circles. The position of the circles comes from
   the most recent messages, one sent locally and the other with a roundtrip to
   the server in between. This makes it easier to visually detect any delays."
  [state]
  (let [local-pos (:local state)
        from-server (:from-server state)]
    [:g
     [:circle (merge circle-defaults {:cx (:x local-pos)
                                      :cy (:y local-pos)})]
     [:circle (merge circle-defaults {:cx (:x from-server)
                                      :cy (:y from-server)
                                      :fill "rgba(0,0,255,0.1)"})]]))
```

This one renders an SVG group with the two circles inside. Then, there are some defaults for the different elements, which can be merged with more specific maps as desired:

```
(def circle-defaults {:fill "rgba(255,0,0,0.1)" :stroke "black" :stroke-width 2 :r 15})
(def text-default {:stroke "none" :fill "black" :style {:font-size 12}})
(def text-bold (merge text-default {:style {:font-weight :bold :font-size 12}}))
```

Finally, there's the `mouse-hist-view` function:

```
(defn mouse-hist-view
  "Render SVG group with filled circles from a vector of mouse positions in state."
  [state state-key stroke fill]
  (let [positions (map-indexed vector (state-key state))]
    (when (seq positions)
      [:g {:opacity 0.5}
       (for [[idx pos] positions]
         ^{:key (str "circle" state-key idx)}
         [:circle {:stroke       stroke
                   :stroke-width 2
                   :r            15
                   :cx           (:x pos)
                   :cy           (:y pos)
                   :fill         fill}])])))
```

Here, the history of mouse movements is rendered, either for your local mouse movements, or the last 1000 from all users. You've seen how that looks like in the screenshot above.

3.2 :server/ptr-cmp

That's it for the rendering of the mouse element. The messages emitted there then get sent both to the client-side and the server-side store components. Let's discuss the server side first, before looking into the wiring of the components. It's really short; this is the entire **example.pointer**[16] namespace:

```
(ns example.pointer
  "This component receives messages, keeps a counter, decorates them with the
   state of the counter, and sends them back. Here, this provides a way to
   measure roundtrip time from the UI, as timestamps are recorded as the message
   flows through the system.
   Also records a recent history of mouse positions for all clients, which the
   component provides to clients upon request.")

(defn process-mouse-pos
  "Handler function for received mouse positions, increments counter and returns
   mouse position to sender."
  [{:keys [current-state msg-meta msg-payload]}]
  (let [new-state (-> current-state
                      (update-in [:count] inc)
                      (update-in [:mouse-moves]
                                 #(vec (take-last 1000 (conj % msg-payload)))))]
```

[16] https://github.com/matthiasn/systems-toolbox/blob/master/examples/trailing-mouse-pointer/src/cljc/example/pointer.cljc

```
    {:new-state new-state
     :emit-msg (with-meta
                 [:mouse/pos (assoc msg-payload :count (:count new-state))]
                 msg-meta)}))

(defn get-mouse-hist
  "Gets the recent mouse position history from server."
  [{:keys [current-state msg-meta]}]
  {:emit-msg (with-meta [:mouse/hist (:mouse-moves current-state)] msg-meta)})

(defn cmp-map
  [cmp-id]
  {:cmp-id      cmp-id
   :state-fn    (fn [_] {:state (atom {:count 0 :mouse-moves []})})
   :handler-map {:mouse/pos      process-mouse-pos
                 :mouse/get-hist get-mouse-hist}
   :opts        {:msgs-on-firehose      true
                 :snapshots-on-firehose true}})
```

At the bottom, you see the `cmp-map`, which again is the map specifying the component that the switchboard will then instantiate. Inside, there's the `:state-fn`, which does nothing but create the initial state inside an atom. Then, there's the `:handler-map`, which here handles the two message types `:cmd/mouse-pos` and `:mouse/get-hist`.

The `process-mouse-pos` handler function then gets the `current-state`, the `msg-payload`, and the `msg-meta` inside the map it gets passed as a single argument, and returns both the `:new-state` and a message to emit, which is the same message it received, only now enriched by the `:count` from this component's state. Note that we are reusing the `msg-meta` from the original message, as this metadata also contains the `:sente-uid` of the client, which is required to route the message back to where it originated. There's more information on the metadata; we'll get to that later. Also, this function maintains the last 1001 positions from all connected client by taking the last 1000 and conjoining the received position.

The `get-mouse-hist` handler function returns the history of mouse moves that's maintained in the `:server/ptr-cmp` back to the client. Once again, the `:sente-uid` on the metadata contains the requester's ID, so we pass on the `msg-meta` in the response.

Next, the messages need to get from the UI component to the server, and back to the client. Here's how that looks like:

[message flow drawing]

3.3 example.core on client side

For establishing these connections, let's have a look at the `core` namespaces on both server and client, starting with the **client**[17]:

[17] https://github.com/matthiasn/systems-toolbox/blob/master/examples/trailing-mouse-pointer/src/cljs/example/core.cljs

```
(ns example.core
  (:require [example.spec]
            [example.store :as store]
            [example.ui-histograms :as hist]
            [example.ui-mouse-moves :as mouse]
            [example.ui-info :as info]
            [example.metrics :as metrics]
            [example.observer :as observer]
            [matthiasn.systems-toolbox.switchboard :as sb]
            [matthiasn.systems-toolbox-sente.client :as sente]))

(enable-console-print!)

(defonce switchboard (sb/component :client/switchboard))

; TODO: maybe firehose messages should implicitly be relayed?
(defn init! []
  (sb/send-mult-cmd
    switchboard
    [[:cmd/init-comp
      #{(sente/cmp-map :client/ws-cmp
                       {:relay-types      #{:mouse/pos
                                            :mouse/get-hist
                                            :firehose/cmp-put
                                            :firehose/cmp-recv
                                            :firehose/cmp-publish-state
                                            :firehose/cmp-recv-state}
                        :msgs-on-firehose true})
        (mouse/cmp-map :client/mouse-cmp)
        (info/cmp-map :client/info-cmp)
        (store/cmp-map :client/store-cmp)
        (hist/cmp-map :client/histogram-cmp)}]
     [:cmd/route {:from :client/mouse-cmp
                  :to   #{:client/store-cmp :client/ws-cmp}}]
     [:cmd/route {:from :client/ws-cmp
                  :to   :client/store-cmp}]
     [:cmd/route {:from :client/info-cmp
                  :to   #{:client/store-cmp :client/ws-cmp}}]
     [:cmd/observe-state {:from :client/store-cmp
                          :to   #{:client/mouse-cmp
                                  :client/histogram-cmp
                                  :client/info-cmp}}]])
  (metrics/init! switchboard)
  (observer/init! switchboard))

(init!)
```

First, as usual, we create a switchboard. Then, we send messages to the switchboard, with the blueprints for the components we want the switchboard to initialize. For the core functionality

discussed so far, only three of them are important: `:client/ws-cmp`, `:client/mouse-cmp`, and `:client/store-cmp`. We'll look at the other components later.

Note that the switchboard is kept in a `defonce`, which means that it can't be redefined later on. This is necessary for working with **Figwheel**[18], as it allows the switchboard to shut down existing components and fire them up again after reload, while retaining the previous component state. Otherwise, without the `defonce`, the old state of each component would be lost as there would be an entirely new switchboard.

Then, inside the component init block, the `:client/ws-cmp` is fired up first. This is the WebSockets component provided by the **systems-toolbox-sente**[19] library. Here, we specify that only messages of the types `:mouse/pos` and `:mouse/get-hist` should be relayed to the server.

Next, we wire the components together:

- messages from `:client/mouse-cmp` are sent to both `:client/store-cmp` and `:client/ws-cmp`
- messages from `:client/ws-cmp` are sent to both `:client/store-cmp` and `:client/jvmstats-cmp`
- messages from `:client/info-cmp` are sent to both `:client/store-cmp` and `:client/ws-cmp`
- `:client/mouse-cmp`, `:client/histogram-cmp` and `:client/info-cmp` all observe the state of the `:client/store-cmp`
- finally, the `:client/observer-cmp` is attached to the firehose, but more about that later when we look at `:client/observer-cmp`.

At the bottom of the namespace, we also fire up the observer and metrics components. We'll look at that when covering the respective components.

3.4 example.core on server side

With the client-side wiring in place, let's look at the server-side wiring in **core.clj**[20]:

```
(ns example.core
  (:require [example.spec]
            [matthiasn.systems-toolbox.switchboard :as sb]
            [matthiasn.systems-toolbox-sente.server :as sente]
            [example.metrics :as metrics]
            [matthiasn.systems-toolbox-observer.probe :as probe]
            [example.index :as index]
            [clojure.tools.logging :as log]
            [clj-pid.core :as pid]
            [example.pointer :as ptr]))

(defonce switchboard (sb/component :server/switchboard))

(defn restart!
```

[18] https://github.com/bhauman/lein-figwheel

[19] https://github.com/matthiasn/systems-toolbox-sente

[20] https://github.com/matthiasn/systems-toolbox/blob/master/examples/trailing-mouse-pointer/src/clj/example/core.clj

```
  "Starts or restarts system by asking switchboard to fire up the provided
   ws-cmp and the ptr component, which handles and counts messages about mouse
   moves."
  []
  (sb/send-mult-cmd
    switchboard
    [[:cmd/init-comp #{(sente/cmp-map :server/ws-cmp index/sente-map)
                       (ptr/cmp-map :server/ptr-cmp)}]
     [:cmd/route {:from :server/ptr-cmp :to :server/ws-cmp}]
     [:cmd/route {:from :server/ws-cmp :to :server/ptr-cmp}]])
  (metrics/start! switchboard)
  #_
  (probe/start! switchboard))

(defn -main
  "Starts the application from command line, saves and logs process ID. The
   system that is fired up when restart! is called proceeds in core.async's
   thread pool. Since we don't want the application to exit when just because
   the current thread is out of work, we just put it to sleep."
  [& args]
  (pid/save "example.pid")
  (pid/delete-on-shutdown! "example.pid")
  (log/info "Application started, PID" (pid/current))
  (restart!)
  (Thread/sleep Long/MAX_VALUE))
```

Here, just like on the client side, a switchboard is kept in a `defonce`. Then, we ask the switchboard to instantiate two components for us, the `:server/ws-cmp` and the `:server/ptr-cmp`, and then wire a simple message flow together.

We've already discussed the `:server/ptr-cmp` above. The `:server/ws-cmp` is the server side of the Sente-WebSockets component, and it takes a configuration map, which you can find in the **example.index**[21] namespace:

```
(def sente-map
  "Configuration map for sente-cmp."
  {:index-page-fn index-page
   :relay-types   #{:mouse/pos :stats/jvm :mouse/hist}})
```

In this configuration map, we tell the component to relay three message types, `:mouse/pos`, `:stats/jvm`, and `:mouse/hist`. Also, we provide a function that renders the static HTML that is served to the clients. Have a look at the namespace to learn more. In particular, watch out for elements with an ID, such as `[:div#mouse]`, `[:figure#histograms.fullwidth]`, `[:div#info]`, or `[:div#observer]`. The client-side application will render dynamic content into these DOM elements.

Then, also in the server-side `example.core` namespace, there is the `-main` function, which is the entry point into the application. Here, we save a PID file, which will contain the process

[21] https://github.com/matthiasn/systems-toolbox/blob/master/examples/trailing-mouse-pointer/src/clj/example/index.clj

ID, also log the PID, and `start!` the application. We also start the server-side portion of the metrics gathering and display, but more about that later.

Finally, we let the main thread sleep until roughly the end of time, or until the application gets killed, whatever happens first. Well, `Long/MAX_VALUE` in milliseconds is only until roughly 292 million years from now, but hey, that should be enough.

3.5 Application Reload from the REPL

Oh, before I forget, you can also reload the server side on the JVM from the **REPL**[22], without long startup times, and while retaining application state. Try this:

```
$ lein repl

example.core=> (restart!)
```

This starts the server side application. Now change something, let's say in the `example.pointer` namespace, for example to print the message payload in `process-mouse-pos`:

```
(defn process-mouse-pos
  "Handler function for received mouse positions, increments counter and returns
   mouse position to sender."
  [{:keys [current-state msg-meta msg-payload]}]
  (let [new-state (-> current-state
                      (update-in [:count] inc)
                      (update-in [:mouse-moves]
                                 #(vec (take-last 1000 (conj % msg-payload)))))]
    {:new-state new-state
     :emit-msg (with-meta
                 [:mouse/pos (assoc msg-payload :count (:count new-state))]
                 msg-meta)}))
```

With this change, all you need to do now is reload the modified namespace, and then call `restart!` again:

```
example.core=> (require '[example.pointer :as ptr] :reload)
example.core=> (restart!)
```

You will see that the application keeps functioning, while maintaining component state, with the only difference that now the message payloads get printed. Magic. Almost as cool as Figwheel, and much better than having to wait ten seconds for the JVM to start up after every change. Note that the sente components don't get reloaded by default because of the `:reload-cmp false` in their config. You can do the same in any of your components where required.

3.6 :client/store-cmp

Okay, now we have the message flow from capturing the mouse events to the server and back. Next, let's look at what happens to those events when they are back at the client. Processing of the returned data happens in the **example.store namespace**[23]:

[22] http://clojure.org/reference/repl_and_main

[23] https://github.com/matthiasn/systems-toolbox/blob/master/examples/trailing-mouse-pointer/src/cljs/example/store.cljs

```
(ns example.store)

(defn mouse-pos-handler
  "Handler function for mouse position messages. When message from server:
    - determine the round trip time (RTT) by subtracting the message creation
      timestamp from the timestamp when the message is finally received by the
      store component.
    - determine server side processing time is determined. For this, we can use
      the timestamps from when the ws-cmp on the server side emits a message
      coming from the client and when the processed message is received back for
      delivery to the client.
    - update component state with the new mouse location under :from-server.
   When message received locally, only update position in :local."
  [{:keys [current-state msg-payload msg-meta]}]
  (let [new-state
        (if (:count msg-payload)
          (let [mouse-out-ts (:out-ts (:client/mouse-cmp msg-meta))
                store-in-ts (:in-ts (:client/store-cmp msg-meta))
                rt-time (- store-in-ts mouse-out-ts)
                srv-ws-meta (:server/ws-cmp msg-meta)
                srv-proc-time (- (:in-ts srv-ws-meta) (:out-ts srv-ws-meta))]
            (-> current-state
                (assoc-in [:from-server] (assoc msg-payload :rt-time rt-time))
                (update-in [:count] inc)
                (update-in [:rtt-times] conj rt-time)
                (update-in [:server-proc-times] conj srv-proc-time)
                (update-in [:network-times] conj (- rt-time srv-proc-time))))
          (-> current-state
              (assoc-in [:local] msg-payload)
              (update-in [:local-hist] conj msg-payload)))]
    {:new-state new-state}))

(defn show-all-handler
  "Toggles boolean value in component state for provided key."
  [{:keys [current-state msg-payload]}]
  {:new-state (update-in current-state [:show-all msg-payload] not)})

(defn mouse-hist-handler
  "Saves the received vector with mouse positions in component state."
  [{:keys [current-state msg-payload]}]
  {:new-state (assoc-in current-state [:server-hist] msg-payload)})

(defn state-fn
  "Return clean initial component state atom."
  [_put-fn]
  {:state (atom {:count             0
                 :rtt-times         []
                 :network-times     []
                 :server-proc-times []
```

```
                :local             {:x 0 :y 0}
                :show-all          {:local  false
                                    :remote false}})})

(defn cmp-map
  "Configuration map that specifies how to instantiate component."
  [cmp-id]
  {:cmp-id      cmp-id
   :state-fn    state-fn
   :handler-map {:mouse/pos    mouse-pos-handler
                 :cmd/show-all show-all-handler
                 :mouse/hist   mouse-hist-handler}
   :opts        {:msgs-on-firehose      true
                 :snapshots-on-firehose true}})
```

The `cmp-map` function once again generates the blueprint for how to instantiate this component. We specify that the initial component state is generated by calling the `state-fn`, which is a map with some keys as you can see above. Then, there are handler functions for three message types `:mouse/pos`, `:cmd/show-all`, and `:mouse/hist`, which we'll look at in detail. Finally, there is some configuration in `:opts`, which specifies that both messages and state snapshots should go on the firehose. We'll discuss the firehose when looking into the `:client/observer` component.

The most important handler function in this application is the `mouse-pos-handler` function. This function receives all `:mouse/pos` messages, which in this application can come either directly from the `:client/mouse-cmp` or from the `:server/ptr-cmp`. Where an individual message comes from is determined by the predicate `(:count msg-payload)` in the if statement. If that key exists, the message comes from the server, otherwise it's directly from `:client/mouse-cmp`.

In case the message is local, we do return new-state altered like this:

```
(-> current-state
    (assoc-in [:local] msg-payload)
    (update-in [:local-hist] conj msg-payload))
```

First, we set the `:local` key to contain the latest mouse position; then we add it to the local history.

The branch when the message comes from the server is slightly more involved:

```
(let [mouse-out-ts (:out-ts (:client/mouse-cmp msg-meta))
      store-in-ts (:in-ts (:client/store-cmp msg-meta))
      rt-time (- store-in-ts mouse-out-ts)
      srv-ws-meta (:server/ws-cmp msg-meta)
      srv-proc-time (- (:in-ts srv-ws-meta) (:out-ts srv-ws-meta))]
  (-> current-state
      (assoc-in [:from-server] (assoc msg-payload :rt-time rt-time))
      (update-in [:count] inc)
      (update-in [:rtt-times] conj rt-time)
      (update-in [:server-proc-times] conj srv-proc-time)
      (update-in [:network-times] conj (- rt-time srv-proc-time))))
```

Here, we calculate a few durations, the `rt-time`, which is the entire roundtrip time, and the `srv-proc-time`, which the duration between the `:server/ws-cmp` passing the message from the client on, and the same component encountering the response. For fully understanding this, you need to know that the **systems-toolbox** automatically timestamps messages when they are received or sent by any component, and saves that on the message metadata.

Here's how the metadata looks like when the `:client/store-cmp` receives a `:mouse/pos` message from the server:

```
{:server/ws-cmp     {:out-ts 1467046063466
                     :in-ts  1467046063467}
 :sente-uid         "25450474-0887-4612-b5ad-07d1ca1f4885"
 :server/ptr-cmp    {:in-ts  1467046063467
                     :out-ts 1467046063467}
 :cmp-seq           [:client/mouse-cmp
                     :client/ws-cmp
                     :server/ptr-cmp
                     :server/ws-cmp
                     :client/store-cmp]
 :client/mouse-cmp {:out-ts 1467046063454}
 :client/store-cmp {:in-ts 1467046063506}
 :client/ws-cmp     {:in-ts  1467046063465
                     :out-ts 1467046063488}
 :tag               "61f2f357-3d12-40ff-9827-8a481cf36f75"
 :corr-id           "a31f12e7-33fb-48a8-833b-3d764c2c14bc"}
```

In contrast, this is how it looks like when the message comes directly from `:client/mouse-cmp`:

```
{:cmp-seq           [:client/mouse-cmp :client/store-cmp]
 :client/mouse-cmp {:out-ts 1467046063476}
 :corr-id           "2d32de55-cf1e-4646-8709-0c02c66d260f"
 :tag               "a7ebdac0-ce78-4e47-adbc-0b955efef5b4"
 :client/store-cmp {:in-ts 1467046063478}}
```

Of course, we could have also looked for the existence of the `:server/ptr-cmp` key on the metadata, rather than looking for the `:count` key on the payload in the branching logic when determining if a message comes from the server, it does not matter.

Okay, back to the `:client/store-cmp`. We do a little bit more there:

```
(update-in [:network-times] conj (- rt-time srv-proc-time)
```

Here, the RTT times are collected in a sequence so we can use the individual values as input to the histograms.

Next, there's the `show-all-handler` function to look at:

```
(defn show-all-handler
  "Toggles boolean value in component state for provided key."
  [{:keys [current-state msg-payload]}]
  {:new-state (update-in current-state [:show-all msg-payload] not)})
```

This handler toggles the value in the view configuration for showing either `:local` or the `:remote` history of mouse positions. These are then used as switches in the `:client/mouse-cmp`, as we've seen above. Finally, there's the `mouse-hist-handler` function:

```
(defn mouse-hist-handler
  "Saves the received vector with mouse positions in component state."
  [{:keys [current-state msg-payload]}]
  {:new-state (assoc-in current-state [:server-hist] msg-payload)})
```

This handler takes care of a sequence of mouse positions received from the server and stores them in the component state, which is returned under the `:new-state` key in the returned map. If these are shown is then dependent on the `:remote` key in the `:show-all` map inside the component state. Typically, when the `:mouse/hist` is received, this switch will be set to true, as the request for these values and switching this key on will have been sent by the `:client/info-cmp` at the same time. The beauty of the UI component watching the state of another component which holds the application state is that we don't have to do anything else. Once the data is back from the server, the mouse component will just know that it needs to re-render itself, now with the new data available. This was all to the `:client/store-cmp`, so let's look into the next component, where the histograms are rendered. But actually, now might be a good time to take a break and go for a walk.

3.7 :client/histogram-cmp

Okay, ready? Let's move on. We've got some ground to cover. The `:client/histogram-cmp` in the **example.ui-histograms namespace**[24] makes use of the data we just collected:

```
(ns example.ui-histograms
  (:require [matthiasn.systems-toolbox-ui.reagent :as r]
            [matthiasn.systems-toolbox-ui.charts.histogram :as h]
            [matthiasn.systems-toolbox-ui.charts.math :as m]))

(defn histograms-view
  "Renders histograms with different data sets, labels and colors."
  [{:keys [observed]}]
  (let [state @observed
        rtt-times (:rtt-times state)
        server-proc-times (:server-proc-times state)
        network-times (:network-times state)]
    [:div
     [:div
      [h/histogram-view rtt-times "Roundtrip t/ms" "#D94B61"]
```

[24] https://github.com/matthiasn/systems-toolbox/blob/master/examples/trailing-mouse-pointer/src/cljs/example/ui_histograms.cljs

```
      [h/histogram-view (m/percentile-range rtt-times 99)
       "Roundtrip t/ms (within 99th percentile)" "#D94B61"]
      [h/histogram-view (m/percentile-range rtt-times 95)
       "Roundtrip t/ms (within 95th percentile)" "#D94B61"]]
     [:div
      [h/histogram-view network-times
       "Network time t/ms (within 99th percentile)" "#66A9A5"]
      [h/histogram-view
       (m/percentile-range network-times 95)
       "Network time t/ms (within 95th percentile)" "#66A9A5"]
      [h/histogram-view server-proc-times
       "Server processing time t/ms" "#F1684D"]]]))

(defn cmp-map
  [cmp-id]
  (r/cmp-map {:cmp-id  cmp-id
              :view-fn histograms-view
              :dom-id  "histograms"
              :cfg     {:throttle-ms           100
                        :msgs-on-firehose      true
                        :snapshots-on-firehose true}}))
```

The most exciting stuff here happens in the histogram namespace of the **systems-toolbox-ui** library, but we'll get there. There are some things of interest here anyway. Did you notice the `:throttle-ms` key in the `:cfg` of the `cmp-map`? This tells the systems-toolbox to deliver new state snapshots only every 100 milliseconds. This throttling is done because it is expensive enough to calculate the histograms for us not to want to do it on every frame. Ten times a second appears to be a good compromise between feeling alive and saving some CPU cycles.

The rest of this namespace is probably not terribly surprising by now. The `histograms-view` function, which is the `:view-fn` of this **systems-toolbox-ui** component, renders a `:div` with six different `histogram-views`, which each renders into an SVG with the chart itself. In some cases, we do some data manipulation first, such as the `hist/percentile-range` from the library namespace. Notice that there are two `:divs` inside the parent, each with three elements inside? That's for the **Flexible Box**[25] layout, also known as **flexbox**. The rest of the layout is then done in **CSS**[26]:

```
#histograms {
    margin-bottom: 1em;
}

#histograms div div{
    display: flex;
    flex-flow: row;
}
```

So what happens here is that we have two `flex` elements, each with `flex-flow: row;` so that each triplet will cover a row inside the available space.

[25] https://www.w3.org/TR/2016/CR-css-flexbox-1-20160526/

[26] https://github.com/matthiasn/systems-toolbox/blob/master/examples/trailing-mouse-pointer/resources/public/css/example.css

Okay, that's it in this namespace.

3.8 matthiasn.systems-toolbox-ui.charts.histogram

The most interesting stuff for rendering the histograms happens in the **matthiasn.systems-toolbox-ui.charts.histogram**[27] namespace. Feel free to skip it if you don't particularly care about constructing **Scalable Vector Graphics**[28]. Also, this is not an introduction to SVG, as that's not the focus of this book, so I will only describe the construction of SVGs with **Reagent**, not what an **SVG** is. If you've never worked with Scalable Vector Graphics, maybe this **tutorial**[29] will give you a gentler introduction.

Okay, with that being said, let's dive into the code:

```
(ns matthiasn.systems-toolbox-ui.charts.histogram
  "Functions for building a histogram, rendered as SVG using Reagent and React."
  (:require [matthiasn.systems-toolbox-ui.charts.math :as m]))

(def text-default {:stroke "none" :fill "black" :style {:font-size 12}})
(def text-bold (merge text-default {:style {:font-weight :bold :font-size 12}}))
(def x-axis-label (merge text-default {:text-anchor :middle}))
(def y-axis-label (merge text-default {:text-anchor :end}))

(defn path
  "Renders path with the given path description attribute."
  [d]
  [:path {:fill         :black
          :stroke       :black
          :stroke-width 1
          :d            d}])

(defn histogram-y-axis
  "Draws y-axis for histogram."
  [x y h mx y-label]
  (let [incr (m/default-increment-fn mx)
        rng (range 0 (inc (m/round-up mx incr)) incr)
        scale (/ h (dec (count rng)))]
    [:g
     [path (str "M" x " " y "l 0 " (* h -1) " z")]
     (for [n rng]
       ^{:key (str "yt" n)}
       [path (str "M" x " " (- y (* (/ n incr) scale)) "l -" 6 " 0")])
     (for [n rng]
       ^{:key (str "yl" n)}
       [:text (merge y-axis-label {:x (- x 10)
                                   :y (- y (* (/ n incr) scale) -4)}) n])
     [:text (let [x-coord (- x 45)
```

[27] https://github.com/matthiasn/systems-toolbox-ui/blob/master/src/cljs/matthiasn/systems_toolbox_ui/charts/histogram.cljs
[28] https://en.wikipedia.org/wiki/Scalable_Vector_Graphics
[29] https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Introduction

```
                y-coord (- y (/ h 3))
                rotate (str "rotate(270 " x-coord " " y-coord ")")]
            (merge x-axis-label text-bold {:x         x-coord
                                           :y         y-coord
                                           :transform rotate})) y-label])))

(defn histogram-x-axis
  "Draws x-axis for histogram."
  [x y mn mx w scale increment x-label]
  (let [rng (range mn (inc mx) increment)]
    [:g
     [path (str "M" x " " y "l" w " 0 z")]
     (for [n rng]
       ^{:key (str "xt" n)}
       [path (str "M" (+ x (* (- n mn) scale)) " " y "l 0 " 6)])
     (for [n rng]
       ^{:key (str "xl" n)}
       [:text (merge x-axis-label {:x (+ x (* (- n mn) scale))
                                   :y (+ y 20)}) n])
     [:text (merge x-axis-label text-bold {:x (+ x (/ w 2))
                                           :y (+ y 48)}) x-label]]))

(defn insufficient-data
  "Renders warning when data insufficient."
  [x y w text]
  [:text {:x           (+ x (/ w 2))
          :y           (- y 50)
          :stroke      "none"
          :fill        "#DDD"
          :text-anchor :middle
          :style       {:font-weight :bold :font-size 24}} text])

(defn histogram-view-fn
  "Renders a histogram. Only takes care of the presentational aspects, the
   calculations are done in the histogram-calc function in
   matthiasn.systems-toolbox-ui.charts.math."
  [{:keys [x y w h x-label y-label color min-bins warning] :as args}]
  (let [{:keys [mn mn2 mx2 rng increment bins binned-freq binned-freq-mx]}
        (m/histogram-calc args)
        x-scale (/ w (- mx2 mn2))
        y-scale (/ (- h 20) binned-freq-mx)
        bar-width (/ (* rng x-scale) bins)]
    [:g
     (if (>= bins min-bins)
       (for [[v f] binned-freq]
         ^{:key (str "bf" x "-" y "-" v "-" f)}
         [:rect {:x      (+ x (* (- mn mn2) x-scale) (* v bar-width))
                 :y      (- y (* f y-scale))
                 :fill   color :stroke "black"
```

```
                  :width  bar-width
                  :height (* f y-scale)}])
       [insufficient-data x y w warning])
     [histogram-x-axis x (+ y 7) mn2 mx2 w x-scale increment x-label]
     [histogram-y-axis (- x 7) y h (or binned-freq-mx 5) y-label]]))

(defn histogram-view
  "Renders an individual histogram for the given data, dimension, label and
   color, with a reasonable size inside a viewBox, which will then scale
   smoothly into any div you put it in."
  [data label color]
  [:svg {:width   "100%"
         :viewBox "0 0 400 250"}
   (histogram-view-fn {:data     data
                       :x        80
                       :y        180
                       :w        300
                       :h        160
                       :x-label  label
                       :y-label  "Frequencies"
                       :warning  "insufficient data"
                       :color    color
                       :bin-cf   0.8
                       :min-bins 5
                       :max-bins 25})])
```

Okay, that was a bit involved. But hey, to use a histogram in your project, all you need is to import this namespace, and then use a one-liner to plot your histogram (and more chart types to come - feel free to contribute).

After skim reading the namespace, are you still interested in constructing charts? Good, then let's go through function by function:

```
(defn histogram-view
  "Renders an individual histogram for the given data, dimension, label and
   color, with a reasonable size inside a viewBox, which will then scale
   smoothly into any div you put it in."
  [data label color]
  [:svg {:width   "100%"
         :viewBox "0 0 400 250"}
   (histogram-view-fn {:data     data
                       :x        80
                       :y        180
                       :w        300
                       :h        160
                       :x-label  label
                       :y-label  "Frequencies"
                       :warning  "insufficient data"
                       :color    color
                       :bin-cf   0.8
```

```
                    :min-bins 5
                    :max-bins 25})])
```

The `histogram-view` function simply creates a container `:svg` element, which scales into its parent element through the `:width "100%"` setting. Also note the `:viewBox "0 0 400 250"`, which allows us to work with a useful coordinate system that's independent of the size of the rendered element. Finally, we pass some data to the `histogram-view-fn`, which we'll look into next.

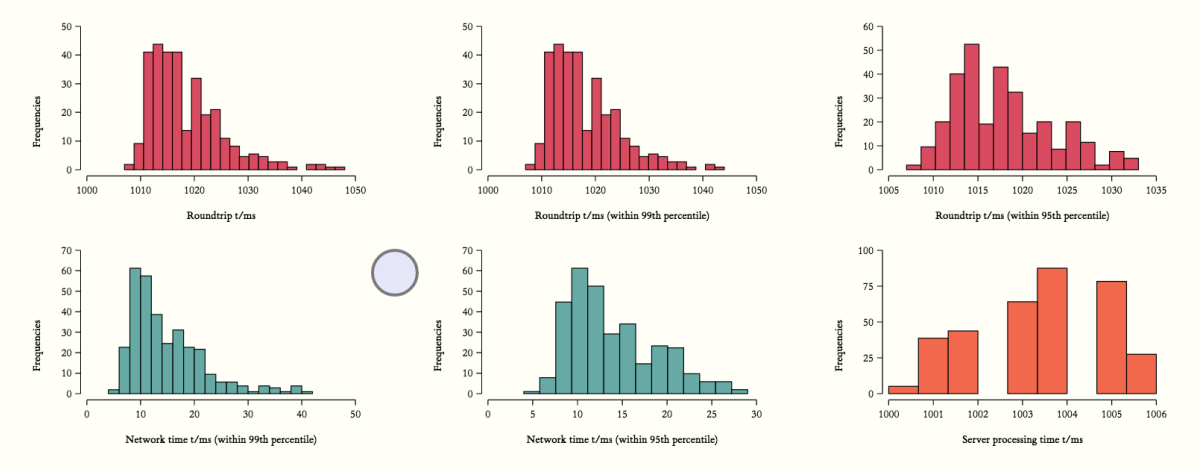
```
(defn histogram-view-fn
  "Renders a histogram. Only takes care of the presentational aspects, the
   calculations are done in the histogram-calc function in
   matthiasn.systems-toolbox-ui.charts.math."
  [{:keys [x y w h x-label y-label color min-bins warning] :as args}]
  (let [{:keys [mn mn2 mx2 rng increment bins binned-freq binned-freq-mx]}
        (m/histogram-calc args)
        x-scale (/ w (- mx2 mn2))
        y-scale (/ (- h 20) binned-freq-mx)
        bar-width (/ (* rng x-scale) bins)]
    [:g
     (if (>= bins min-bins)
       (for [[v f] binned-freq]
         ^{:key (str "bf" x "-" y "-" v "-" f)}
         [:rect {:x      (+ x (* (- mn mn2) x-scale) (* v bar-width))
                 :y      (- y (* f y-scale))
                 :fill   color :stroke "black"
                 :width  bar-width
                 :height (* f y-scale)}])
       [insufficient-data x y w warning])
     [histogram-x-axis x (+ y 7) mn2 mx2 w x-scale increment x-label]
     [histogram-y-axis (- x 7) y h (or binned-freq-mx 5) y-label]]))
```

Above, we render an **SVG g element**[30], which contains the histogram. Before we can render the bars of the histogram, we need to calculate a few things from the provided data, which happens in the first line in the `let` binding:

`{:keys [mn mn2 mx2 rng increment bins binned-freq binned-freq-mx]} (m/histogram-calc args)`

We will look into the calculations in the next section. For the dicussion here, we only need to know what each one does. `mn` is the minimum value in the data. `m2` is the minimum rounded down to the next increment, as it's not always useful to start the x-axis from zero. Let me show you an example:

[30] https://developer.mozilla.org/en/docs/Web/SVG/Element/g



1000 ms delay

Here, all our values are between 1000 and 1050 milliseconds. The histogram would be fairly useless if the x-axis started at zero because the bars would be so thin that we probably couldn't even see them, let alone tell apart.

We will look at creating the delay of 1000ms for each message in a subsequent chapter when discussing the **systems-toolbox** messaging model.

Next, we have `rng`, which is the distance between the minimum and the maximum value. Then, there's the `increment`. This is the distance between the ticks on the x-axis, such as `10`, `25`, or also `500`, depending on the range of the provided `data`. `bins` is the number of bins in the histogram, each of which will be represented as a bar. `binned-freq` is contains the frequencies per bin. Finally, `binned-freq-mx` is the maximum frequency in any of the bins, which is used to determine the scale on the y-axis.

With those values, we can calculate the `x-scale` and `y-scale`, which will be used to translate positions into the given coordinate system. Finally, we can determine the width of each bar, by dividing the product of `rng` and `x-scale` through the number of `bins`.

With those calculations completed, we can render the histogram into a `:g` element. Here, the bars are only displayed if there are enough bins. Otherwise, we display `"insufficient data"`. The number of bins is configured in the `:min-bins` key of the argument map. When called from the `histogram-view`, I've chosen a minimum of five bins. This value is entirely arbitrary but seems to work fairly well. Less than five bins look stupid and don't provide much meaningful information either.

If the data is deemed sufficient, we render a vertical bar as a `:rect` for each bin. This rendering happens in a **for-comprehension**[31], as you've already seen in the previous chapter. Of importance here is the `:key` on each elements' metadata. While we would get by without, **React** needs this key to work more efficiently by reusing elements in the next render cycle. Without assigning the keys, the browser needs to do more work and React prints long and ugly warnings in the browser's console.

Then, we render the **x-axis** by calling `histogram-x-axis`, and the **y-axis** in `histogram-y-axis`.

[31] https://clojuredocs.org/clojure.core/for

The functions for rendering the axes are fairly straightforward. Here's the `histogram-x-axis` function:

```
(defn histogram-x-axis
  "Draws x-axis for histogram."
  [x y mn mx w scale increment x-label]
  (let [rng (range mn (inc mx) increment)]
    [:g
     [path (str "M" x " " y "l" w " 0 z")]
     (for [n rng]
       ^{:key (str "xt" n)}
       [path (str "M" (+ x (* (- n mn) scale)) " " y "l 0 " 6)])
     (for [n rng]
       ^{:key (str "xl" n)}
       [:text (merge x-axis-label {:x (+ x (* (- n mn) scale))
                                   :y (+ y 20)}) n])
     [:text (merge x-axis-label text-bold {:x (+ x (/ w 2))
                                           :y (+ y 48)}) x-label]]))
```

The mechanism here will probably look fairly familiar by now:

- there's a group inside the `:g` element
- next, there's the axis itself, rendered by the `path` function
- there's a `for`-comprehension for the ticks on the axis, which also use the `path` function
- there's another `for`-comprehension for the axis labels (the numbers associated with a tick)
- finally, there's a label, which in our example application here would, for example, be `"Roundtrip t/ms"`

Both `for`-comprehensions make use of the range `rng`, which is a sequence from `mn` to one larger than `mx`, with the step size `increment`. All these values depend on the data and are computed individually, as we will see in the next section.

Here's the aforementioned `path` function, which is only a thin wrapper over `:path`, with a few defaults, so we save some typing later on:

```
(defn path
  "Renders path with the given path description attribute."
  [d]
  [:path {:fill         :black
          :stroke       :black
          :stroke-width 1
          :d            d}])
```

The `histogram-y-axis` is somewhat similar, only that here we can calculate more in the function, as we don't need `scale` or `rng` in the calculation of the bins:

```
(defn histogram-y-axis
  "Draws y-axis for histogram."
  [x y h mx y-label]
  (let [incr (m/default-increment-fn mx)
        rng (range 0 (inc (m/round-up mx incr)) incr)
        scale (/ h (dec (count rng)))]
    [:g
     [path (str "M" x " " y "l 0 " (* h -1) " z")]
     (for [n rng]
       ^{:key (str "yt" n)}
       [path (str "M" x " " (- y (* (/ n incr) scale)) "l -" 6 " 0")])
     (for [n rng]
       ^{:key (str "yl" n)}
       [:text (merge y-axis-label {:x (- x 10)
                                   :y (- y (* (/ n incr) scale) -4)}) n])
     [:text (let [x-coord (- x 45)
                  y-coord (- y (/ h 3))
                  rotate (str "rotate(270 " x-coord " " y-coord ")")]
              (merge x-axis-label text-bold {:x         x-coord
                                             :y         y-coord
                                             :transform rotate})) y-label]]))
```

Other than calculating the `rng` and `scale` locally inside the `let`-binding, the function is pretty much the same as the `histogram-y-axis` function, with the other difference that the paths and labels are rotated, as obviously, the y-axis is vertical. If you want to learn more about **SVG** paths, I'd recommend either one of the tutorials out there, or to just modify the values and see how that affects the histogram. For that, I would copy the code over into the sample app and use **Figwheel** for immediate feedback. Otherwise, you'd have to publish the library locally after each change, and then recompile the sample application, which takes away all the fun. Tight feedback loops are important.

This is all for the rendering of the histogram. In the next chapter, I'll guide you through the math behind the calculations used here.

If you want to use a histogram in your application and are happy with the defaults, you can simply call the `histogram-view` function. Or, if you want more fine-grained control, you can copy this function and use the values you desire. Or, you can, of course, use this whole thing as an inspiration and come up with your own chart. In that case, please consider submitting a PR, others might benefit from that, too.

Questions? Send me an email; I'm happy to help. matthias.nehlsen@gmail.com

3.9 matthiasn.systems-toolbox-ui.charts.math

In the `matthiasn.systems-toolbox-ui.charts.histogram` namespace, we used a handful of mathematical helper functions. These live in the **matthiasn.systems-toolbox-ui.charts.math**[32] namespace. This is actually a **cljc**[33] file, which allows us to target both **Clojure** and **Clojure-**

[32] https://github.com/matthiasn/systems-toolbox-ui/blob/master/src/cljc/matthiasn/systems_toolbox_ui/charts/math.cljc

[33] https://github.com/clojure/clojurescript/wiki/Using-cljc

Script. This is very useful for **testing** the functions on the **JVM**, which I much prefer over testing in the browser. Here's the entire namespace:

```
(ns matthiasn.systems-toolbox-ui.charts.math)

(defn mean
  "From: https://github.com/clojure-cookbook/"
  [coll]
  (let [sum (apply + coll)
        count (count coll)]
    (if (pos? count)
      (/ sum count)
      0)))

(defn median
  "Modified from: https://github.com/clojure-cookbook/
   Adapted to return nil when collection empty."
  [coll]
  (let [sorted (sort coll)
        cnt (count sorted)
        halfway (quot cnt 2)]
    (if (empty? coll)
      nil
      (if (odd? cnt)
        (nth sorted halfway)
        (let [bottom (dec halfway)
              bottom-val (nth sorted bottom)
              top-val (nth sorted halfway)]
          (mean [bottom-val top-val]))))))

(defn interquartile-range
  "Determines the interquartile range of values in a sequence of numbers.
   Returns nil when sequence empty or only contains a single entry."
  [sample]
  (let [sorted (sort sample)
        cnt (count sorted)
        half-cnt (quot cnt 2)
        q1 (median (take half-cnt sorted))
        q3 (median (take-last half-cnt sorted))]
    (when (and q3 q1) (- q3 q1))))

(defn percentile-range
  "Returns only the values within the given percentile range."
  [sample percentile]
  (let [sorted (sort sample)
        cnt (count sorted)
        keep-n (Math/ceil (* cnt (/ percentile 100)))]
    (take keep-n sorted)))
```

```
(defn freedman-diaconis-rule
  "Implements approximation of the Freedman-Diaconis rule for determing bin size
   in histograms: bin size = 2 IQR(x) n^-1/3 where IQR(x) is the interquartile
   range of the data and n is the number of observations in sample x. Argument
   is expected to be a sequence of numbers."
  [sample]
  (let [n (count sample)]
    (when (pos? n)
      (* 2 (interquartile-range sample) (Math/pow n (/ -1 3))))))

(defn round-up [n increment] (* (Math/ceil (/ n increment)) increment))
(defn round-down [n increment] (* (Math/floor (/ n increment)) increment))

(defn best-increment-fn
  "Takes a seq of increments, a desired number of intervals in histogram axis,
   and the range of the values in the histogram. Sorts the values in increments
   by dividing the range by each to determine number of intervals with this
   value, subtracting the desired number of intervals, and then returning the
   increment with the smallest delta."
  [increments desired-n rng]
  (first (sort-by #(Math/abs (- (/ rng %) desired-n)) increments)))

(defn default-increment-fn
  "Determines the increment between intervals in histogram axis.
   Defaults to increments in a range between 1 and 5,000,000."
  [rng]
  (if rng
    (let [multipliers (map #(Math/pow 10 %) (range 0 6))
          increments (flatten (map (fn [i] (map #(* i %) multipliers))
                                   [1 2.5 5]))
          best-increment (best-increment-fn increments 5 rng)]
      (if (zero? (mod best-increment 1))
        (int best-increment)
        best-increment))
    1))

(defn histogram-calc
  "Calculations for histogram."
  [{:keys [data bin-cf max-bins increment-fn]}]
  (let [mx (apply max data)
        mn (apply min data)
        rng (- mx mn)
        increment-fn (or increment-fn default-increment-fn)
        increment (increment-fn rng)
        bin-size (max (/ rng max-bins) (* (freedman-diaconis-rule data) bin-cf))
        binned-freq (frequencies (map (fn [n] (Math/floor (/ (- n mn) bin-size)))
                                      data))]
    {:mn             mn
     :mn2            (round-down (or mn 0) increment)
```

```
    :mx2            (round-up (or mx 10) increment)
    :rng            rng
    :increment      increment
    :binned-freq    binned-freq
    :binned-freq-mx (apply max (map (fn [[_ f]] f) binned-freq))
    :bins           (inc (apply max (map (fn [[v _]] v) binned-freq)))}))
```

The first two functions here, `mean` and `median`, are borrowed from the **Clojure Cookbook**[34]. I've adapted `median` to return `nil` when the collection is empty, as that's useful further on. I've taken those two functions because they are useful in my implementation of the **interquartile range**[35]:

```
(defn interquartile-range
  "Determines the interquartile range of values in a sequence of numbers.
   Returns nil when sequence empty or only contains a single entry."
  [sample]
  (let [sorted (sort sample)
        cnt (count sorted)
        half-cnt (quot cnt 2)
        q1 (median (take half-cnt sorted))
        q3 (median (take-last half-cnt sorted))]
    (when (and q3 q1) (- q3 q1))))
```

The `interquartile-range` function takes a sample, which is a sequence of numbers. If this sequence is empty, `nil` is returned. Otherwise, we `sort` the sequence, `count` it, and then take the `half-cnt`, which is the `floor` of dividing the `cnt` by two. This is the number of items in half the data, minus the halfway point if there is one (when `cnt` is odd). Then the values `q1` and `q3` are defined as the `median` of the first or last half of the values, respectively. Finally, the IQR is returned, which is the difference between `q1` and `q3`, and thus the range of half the data. The interquartile range is something we need to determine when computing the bin size via the **Freedman-Diaconis Rule**[36]:

```
(defn freedman-diaconis-rule
  "Implements approximation of the Freedman-Diaconis rule for determing bin size
   in histograms: bin size = 2 IQR(x) n^-1/3 where IQR(x) is the interquartile
   range of the data and n is the number of observations in sample x. Argument
   is expected to be a sequence of numbers."
  [sample]
  (let [n (count sample)]
    (when (pos? n)
      (* 2 (interquartile-range sample) (Math/pow n (/ -1 3))))))
```

The **Freedman-Diaconis rule**[37] is fairly simply, once we have implemented the IQR:

- determine the IQR

[34] https://github.com/clojure-cookbook/"

[35] https://en.wikipedia.org/wiki/Interquartile_range

[36] https://en.wikipedia.org/wiki/Freedman%E2%80%93Diaconis_rule

[37] https://en.wikipedia.org/wiki/Freedman%E2%80%93Diaconis_rule

- multiply it by 2
- multiply it by the cube root of n, the count of items

Following these steps gives us a suggested size of the bins in a histogram, which can then be used to determine the number of bins and thus the number of bars to display in our histogram.

Then, there's also the `percentile-range` function:

```
(defn percentile-range
  "Returns only the values within the given percentile range."
  [sample percentile]
  (let [sorted (sort sample)
        cnt (count sorted)
        keep-n (Math/ceil (* cnt (/ percentile 100)))]
    (take keep-n sorted)))
```

This function helps when trying to get rid of outliers, which may or may not be helpful in your data. Here, it sometimes helps, for example when all values are in the low hundreds, and there's a single outlier in the thousands, as that outlier would otherwise lead to bins that are too large, with many empty bins. As with all visualization, it depends on the data and requires some experimentation.

Next, there are helpers for determining the intervals at which to put the ticks in the histogram axes:

```
(defn best-increment-fn
  "Takes a seq of increments, a desired number of intervals in histogram axis,
   and the range of the values in the histogram. Sorts the values in increments
   by dividing the range by each to determine number of intervals with this
   value, subtracting the desired number of intervals, and then returning the
   increment with the smallest delta."
  [increments desired-n rng]
  (first (sort-by #(Math/abs (- (/ rng %) desired-n)) increments)))

(defn default-increment-fn
  "Determines the increment between intervals in histogram axis.
   Defaults to increments in a range between 1 and 5,000,000."
  [rng]
  (if rng
    (let [multipliers (map #(Math/pow 10 %) (range 0 6))
          increments (flatten (map (fn [i] (map #(* i %) multipliers))
                                   [1 2.5 5]))
          best-increment (best-increment-fn increments 5 rng)]
      (if (zero? (mod best-increment 1))
        (int best-increment)
        best-increment))
    1))
```

This is an interesting problem. Of course, we could hardwire the increments between the ticks, but then the histogram would hardly be reusable. My initial approach was something this:

```
(defn default-increment-fn
  [rng]
  (cond (> rng 20000) 5000
        (> rng 8000) 2000
        (> rng 3000) 1000
        (> rng 1500) 500
        (> rng 900) 200
        (> rng 400) 100
        (> rng 200) 50
        (> rng 90) 20
        :else 10))
```

Depending on the range `rng`, I would select different spacing between the ticks on an axis. But that's not general enough. So what I came up with instead is this:

- generate some multipliers, such as `(1.0 10.0 100.0 1000.0 10000.0 100000.0)`
- multiply each with `1`, `2.5` and `5`, then flatten those result vectors
- then, with this sequence of possible increments, and a target value of five ticks (which look good in a histogram IMHO), call the `best-increment` function
- there, the candidate increments are sorted by the delta between the desired number of ticks and the number of ticks we'd get with the respective increment
- the first of these sorted values is returned, which is the one with the smallest delta

This approach is much more generic and seems to work well.

I'm always amazed that we can do all these calculations whenever there's a change in the data. The browser has become a powerful environment these days indeed.

Finally in this namespace, there's the `histogram-calc` function:

```
(defn histogram-calc
  "Calculations for histogram."
  [{:keys [data bin-cf max-bins increment-fn]}]
  (let [mx (apply max data)
        mn (apply min data)
        rng (- mx mn)
        increment-fn (or increment-fn default-increment-fn)
        increment (increment-fn rng)
        bin-size (max (/ rng max-bins) (* (freedman-diaconis-rule data) bin-cf))
        binned-freq (frequencies (map (fn [n] (Math/floor (/ (- n mn) bin-size)))
                                      data))]
    {:mn             mn
     :mn2            (round-down (or mn 0) increment)
     :mx2            (round-up (or mx 10) increment)
     :rng            rng
     :increment      increment
     :binned-freq    binned-freq
     :binned-freq-mx (apply max (map (fn [[_ f]] f) binned-freq))
     :bins           (inc (apply max (map (fn [[v _]] v) binned-freq)))}))
```

This function does all the required calculations to get the data into the shape that's required for the actual rendering of the histogram:

- find min value `mn`, max value `mx`, and range of the data `rng`
- calculate `increment` between ticks on the x-axis (note that you can specify your own function for finding the increments here)
- determine size of the bins `bin-size`
- put values into bins in `binned-freq`
- find max frequency in bins (for scaling)
- find number of bins (including empty ones)

Okay, that's all in this namespace. While the material convered here is not strictly related to the rest of the book, I hope you found it interesting nonetheless. Also, we will use the histograms later in the book when looking into observability of systems, and it never hurts to understand your tools a little better.