Invention, Innovation and ClojureScript

- **Speaker: David Nolen**
- **Conference: EuroClojure - June 2014**
- **Video: https://vimeo.com/100574226**

My talk is called Invention, Innovation and ClojureScript. That's a curious title. Some of you may know, I recently left The New York Times. I had a lot of fun there, but I now work for Cognitect working as administering things definitely doing some ClojureScript related things there. Cool stuff. But I want to talk about Invention, Innovation and ClojureScript and to sort of kick that off, if you haven't seen this talk by Alan Kay, the future doesn't have to be incremental. I highly recommend it and he sort of dissects the difference between invention and innovation. He points out like a lot of the work that he was doing with many other people at Xerox PARC, was a sort of a phase in invention in the 70s. They worked for a decade.

In that decade at PARC, they came up with a personal computer, graphical user interfaces, WYSIWYG, a form of OOP that most people would recognize, bitmap screens, laser printers, networking, Ethernet-based networking. It's really cool stuff, but it took them ten years, right? Ten years to come up with this stuff and that's in a scale that, I think, most of us engineers that are involved in businesses, that's probably a little bit too much time. He talks about that innovation is also extremely important. Innovation is the process in which you take these inventions that these tons of research money went into and you figure out, "How do we make this palatable to the market or understandable by normal people?" It really is a lot of work. Innovating which is taking inventions and sort of bringing them to a wider audience.

So to create a little bit more context for this, I highly recommend this book. I recently read this, I think, last year. The Dream Machine was recommended by Alan Kay and it's a history of how interactive computing can be. It uses, sort of, has a J.C.R. Licklider who leaded ARPA in the 60s or for a few years, as a sort of essential character of focus. It's amazing. If you do anything with computers whether it's networking, whether it's computer graphics or interactive computing, servers in the map. This book is incredible. It will blow your mind. Really good stuff.

Again, J.C.R. Licklider, he was a psychologist who became involved in computing. He deeply believed that human factors will play in a very important role, but his big idea was that, in the future, everybody would have their computer. We would all be networked. It would sort of augment our abilities to think. More or less, he helped create the future that we live in today. I mean, he helped ARPA, for example, finance Xerox PARC Research. He also financed some people in this photo. So there's John McCarthy, the inventor of Lisp. Pretty cool. And to his right is a person which some people might not be familiar with, that's Ed Fredkin. Ed Fredkin was a Caltech dropout. He joined the United States Air Force as a fighter pilot for some years and when he got out, it turns out he was an incredible hardware hacker. He was the guy that was able to take this under-powered hardware and sort of build the very first interactive computing systems. He also invented the Trie. Some people in functional programming community like to say /trī/ but it actually is pronounced /trē/ for retrieval, and he invented that data structure which turns out is extremely important.

This is J.C.R. Licklider, one of his most famous essays is, The Man-Computer Symbiosis. This is 54 years old and in this essay, he lays out all the different bits of technology they believe as technological problems that had to be solved in order for interactive computing to happen. And he says, I'm going to just read it out, "The Trie memory scheme is inefficient for small memories, but it becomes increasingly efficient in using available storage space as memory size increases. The attractive features of the scheme are these: One, the retrieval process is extremely simple. Given the argument, enter the standard initial register with the first character, pick up the address of the second. Then go to the second register, and pick up the address of the third, etc. If two arguments have initial characters in common, they use the same storage space for those characters." So if you have ever seen anybody talk about Clojure's particular implementation of persistent data structures, you understand the Tries are at the foundation of their design both for efficiency and space and in time. So that's pretty cool, invented a very long time ago and still very relevant to us today.

So invention is extremely hard. Again, I actually don't think... I mean, invention is extremely critical but at the same time, I think innovation is equally important and honestly, for most of us, that's what we can do. We have the time and the ability to innovate. To be an inventor often just takes far too much money and far too much time but there are a lot of incredible things to solve when you're innovating. What I just shared to you was Douglas Engelbart's original mouse which, you know, wasn't very usable. I think they broke every four to five days. There was a ton of work going into making it more natural for more users and more durable. Again, this is the process of innovation. Taking a truly groundbreaking idea and actually making it work in the real world and I think that this process is extremely important and something to really think about as Clojure and ClojureScript programmers.

We're going to dig a bit more to that. One concrete example of this is purely functional data structures which you probably hear a lot of people cite. So if you've read this book, it is a great book. It's very interesting but the book is about more or less paper complexity, right? It's stuff that looks good on paper and it was sort of, taking a bunch of academic research and say, "Here's some stuff. Let's lay on the foundation in which other people can build variants on this and perhaps things that have better characteristics in the real world." Somebody that did that was Rich Hickey. He often doesn't get credit for inventing the The Bitmap Vector Trie. People think that that came from purely functional data structures. It did not. That was actually an innovation on top of the hash array mapped tree by Phil Bagwell. So again, thinking a little bit more about what can we, as engineers, what can we innovate?

Okay, so keep that in your mind for a second and now I want to talk a bit about the state of ClojureScript. It's been some time. Rich Hickey released this July 20, 2011. So we're just a few weeks shy of three years. I was there. It was very exciting. Those tons of people were there. It was very crowded. Some people were very skeptical three years ago that anything would come of this and I would say, "You know, give it some time and we'll see." A lot has happened since that time. Early on, relevant to the time releasing what they called ClojureScript 1 which is sort of demonstrating that ClojureScript could be used to build scalable softwares. It was a very interesting, sort of, first attempt. Then, you have things like himera which fogus put together which was like a REPL online and then I worked with him and we put together something called translations from JavaScript which is sort of a play on what Google was doing with Dart. They sort of restrained the semantic differences between JavaScript and ClojureScript and a lot of people responded to this. A lot of people we're like, "Oh, come on, another compile to JavaScript language." And when they actually went to this list, they're like, "Actually, that's really compelling. It's a very compelling story when you see what value ClojureScript is adding over something like writing JavaScript by hand."

Okay, but today... So ClojureScript is actually quite popular. It's starred as many times as Clojure on GitHub which is pretty cool, 3200 stars. Not bad for a compile to JavaScript language if you follow that stuff. Even more important is that is has now 81 contributors and this is very good for an open-source project that's three years old. The number of patches that are coming is incredible. The reason that we don't have copy-on-write data structures is because somebody contributed very good ports of the Java implementations directly into ClojureScript. The reason we have source maps is because some contributors spent the time to take some initial work that I'd done and just put a bow on it and wrap it up and make it work. It's been great. There's tons of community contribution. It's increasing. This is awesome. I think ClojureScript definitely has a bright future as an open-source project.

We've also been lucky. There's been several high profile projects I'm sure that you've all heard of LightTable. LightTable is 11,000 or 12,000 lines of ClojureScript, definitely one of the larger ClojureScript applications that I'm aware of. Very cool stuff. Also, there are tons of people now are using it. I'm sure there are many of you in this room that are using it now. I think it's reached the point where people are willing to adopt it because it's pretty stable and it offers a lot of value. So this is a recent post by Prismatic which they're using Om. They've been using ClojureScript, but they were sort building their system off of things that did not try to fix the problems around immutable DOM, so they went from like a 25,000 line ClojureScript program down to a 5,000 line ClojureScript program. And so, what this means is that even though we have something as incredible as ClojureScript, there's a lot of opportunities still to innovate and to provide simpler solutions towards running the type of software that we want to write.

So yeah, tons of things had happened. If you'd been following along, most of you may not have… I don't know. I don't know how many people remember the early days of ClojureScript but we started off with copy-on-write data structures. We started off with really slow incremental compilation times. No source maps, people would like look at the tea leaves of the generated code. So things are really improved since then, but also, the world has not stopped. The world has kept moving, specifically, it's great that we had to write ClojureScript and we target a host, and these hosts really, with an 's,' that are continuing to improve their own systesm. So back three years ago when we switched to persistent data structures, it was a just a basic performance win on most engines, right? Copy-on-write, doesn't really work after like, you know, once you're talking about a hundred items and some collection, so we switched to persistent data structures because even in a basic sense, it would be faster than copy-on-write; however, what was fascinating two years ago, I guess, was that V8 performed very, very, very well. It was not unusual to see performance within two and one-half X of the JVM which is pretty wild. And at the time, we went down this road because our hunch was that the rest of the world's going to catch up. V8 might have a lead, it's not going to last very long. And that has more or less played out.

WebKit, now, has a very strong compiler team. They recently announced something called the FTLJIT, it's a Fourth Tier. They actually used LLVM to get even more optimization, lower level optimizations out of JavaScript core and this is an attempt to get the performance that Firefox is getting out of asm.js without resorting to AOT. WebKit is trying to get dynamic compilation and yet deliver asm.js level performance. This type of thing is great for ClojureScript, right? In ClojureScript, ClojureScript is, the code base is 50% persistent data structure implementation so there's just tons of low level bitshifting or array-accesses. All this stuff around optimization, around asm.js is going to be good for ClojureScript as long as people are pursuing dynamic compilation techniques. I think mostly, a lot of benefit and it seems that WebKit has definitely thrown down the gauntlet and it's going to be exciting to see what the other engines do.

Another extreme development is that Java 8 came out and Java now has this thing called Nashorn which is a high-performance JavaScript engine which is going to replace Rhino. Rhino is end-of-lifed and this is also a very interesting development because ClojureScript, you know, there's still a question of 'how do you run the same code?' In some instances, it might be useful to be able to run ClojureScript that you generated in your JVM process.

So I just want to show a couple of quick demos. I think people have used ClojureScript but I think I just want to show that, like, how good is ClojureScript really at some basic things. I wrote a library called 'mori' which is for JavaScript developers but it's not really important, I'm just using this just to demo the performance of persistent data structures. So here, I'm using a recent build of V8 and what I'm going to do here is I'm comparing the performance of opt adding 1 million items to a mutable JavaScript array and adding 1 million items to an immutable vector under a recent build of V8. Average is 85 milliseconds, average is 230 milliseconds so it's within 3x. We're not even talking … it's not an order of magnitude slower. For many operations, this is already very, very, very good. Definitely when you're talking about client-side applications and you're talking about domain information, this is not going to be your bottleneck. So that's very exciting. But of course, Rich had another bit of innovation here which is very exciting which is he noticed that often you want to build some value and you just want to build it quickly, and you want to relieve GC pressure, you don't really care, you know, you have something mutable for a very short amount of time when you're constructing that value.

We actually have transients in ClojureScript so let's see how they perform. Building transients are faster than building mutable arrays. You can build a transient value and convert it into a persistent value faster than you can build a mutable array under V8 so that's pretty crazy. And just to show that it's not just V8, this is a very recent build of JavaScript core. So yeah, right? The average is 28.1 milliseconds, the average for transient vector 29.8. We're not … This is the type … We're right there. Right there, we can build something which has many more properties than immutable array when we can build them just as fast. Just to show you that

Nashorn is actually pretty interesting technology as well, I want to share a quick demo. This is a less trivial benchmark. This is React, running into command line with Om, so I'm building a very trivial template and I'm benchmarking, you know, the entire react pipeline and Om to build a simple template.

This is building a template a hundred times so it takes 13 milliseconds with V8, recent build of JavaScript core. It's even faster and SpiderMonkey Generator is the slowest of the bunch but it's not too shabby. Let's see Nashorn, so Nashorn's very interesting. I think the court release, it's pretty good. It's going to get better. I believe the next release is going to be an incremental JDK 8 release. It's going to be a lot better but I'm going to show you that it's actually, surprisingly good, way faster than Rhino. So the main issue right now it has slow load time and it has ... the warm up time is long, so these are the two things that are going to get addressed. You'll see there's like three phases or something but once Nashorn is like warmed up, it gets pretty good.

So you can see, so there, right there. You can see that it takes a long time to warm up but when it gets done, you start approaching the perf of SpiderMonkey and actually, the Nashorn team are not trying to beat SpiderMonkey. Their goal is to compete with V8. It might take some time but this is going to happen. It's going to happen in the near future.

Okay. So now what? We have cool technology. We have something actually that not a lot of other compiled to JavaScript languages have, right. Things like TypeScript, things like Dart, these are all under the assumption that people want to build the same ... I mean, in my opinion, the same broken type of stuff we've been building more or less for 35 years. ClojureScript is basically saying, "No." We can build the same type of systems and this is sort of ... I think, the paraphrase Rich Hickey was that, "We can build the same things we've been building or we can build them radically simpler. A lot of the other compile to JavaScript languages just want to continue the status quo. They're not going to change anything at all. I think that in ClojureScript, we have a unique opportunity to rethink how we build these systems that we want to build.

One huge thing that's happened is in the Clojurescript community, definitely is the appearance of React. React is a Facebook project that was announced last year and at first glance, it doesn't look very interesting. Upon a second glance, you might start seeing that under the hood, it was designed with a set of sort of functional mindset and really, the coolest thing about it is that unlike, other JavaScript frameworks which have this deep notion that everything is mutable. I mean, if you use any of these things whether it's Angular, whether it's Backbone, whether it's Ember, mutability is just baked into the thing and you can't fix it. And so, in many sense in the past two years, the ClojureScript community has been struggling with the fact that, 'We want to build these better systems but on what?' Right? There's no existing libraries that make this any easier and certainly what's in the browser is not making it easier at all, so React was kind of like in some sense, was a great appearance because it gave us a way to write idiomatic ClojureScript Programs and now, these immutable data structures that we have that are fast and efficient actually have a lot of power when coupled with the React approach.

Just to summarize really quickly for those who haven't dug into React that much, React basically let's you say, "I have an immutable value. That's my application state. I can apply some functions and I'm going to get a new value which represents the DOM." They call it the virtual DOM. It's just some data structure that represents the DOM.

You can, of course, construct a new value which represents the new app state. We're not going to destroy anything. We have a new app state. The old app state is still there. We apply the function and we get a new value which represents the new representation of the DOM. So what React does is clever and it, again, fits very well with ClojureScript, is that what they do is they use those two values that represents the DOM to compute a minimal change set. And this is fantastic because what this means is that, there's a huge opportunity here for ClojureScript which I spend some time demonstrating other people are also demonstrating is that given this managing state becomes a lot simpler.

We already have a way to talk about app state as an immutable value but now, React gives us a way to say, well, we want to go back in time. We want to recover from a server error. We have a modal and we don't have to manually reconstruct the state. We want to build undo, redo, anything that involves any complex manipulation of state. We effectively get it for free. We can just give React our data structures and React will do the right thing. More or less, what's happened because of this is is that React has completely taken over in the ClojureScript world. So I've created Om like six months ago now, some people use it, but there are other things that are also able to leverage the sort of power of React quite well. There's Reagent, it's very cool. We have Quiescent which is a much sort of thinner layer over React and you have other things that are less known that are still interesting like [inaudible 22:19]. This is great. I don't think that we should stop. Like Om was just one perspective and I only wanted to solve one tiny thing, and demonstrate one small thing in which nobody else was trying even though they probably have really good properties which is that representing an application state as a global and mutable value is something that people aren't looking that enough even though I really was sort of wanting to do that with databases and most databases are global state, and obviously that's not a bad design since most applications work that way, so Om was an experiment to show, "Well, if you do that when we make it immutable, a lot of things that you think were difficult aren't difficult anymore." It's just to demonstrate that. It doesn't mean it's the end. Once that sinks in, I think that people should be looking at what other types of innovations can we make around sources of complexity in our application.

Again, just to drive that home, definitely with Om, the idea was never to ... the idea is not ... we're not going to like make interfaces that nobody's ever seen before. That's not the point. The point is, again, is to build the same type of systems but to build them ... make them more scalable, make them simpler, make them easier to reason about. My favorite thing about the Prismatic blog post when they switched to Om which, again, has nothing to do with Om, is that when they chose an immutable global state design, having a new person come in and be able to reason about the state of the app, having components that don't interact in really crazy ways with the rest of the application, it allows people to work without having to know the details of the entire system which is generally not the case when you're doing sophisticated UI development. So, again, this is about simplifying the development process not making new types of things or new types of User Interface you've never seen before.

Another great example of this that I can show, a concrete example is goya which some of you may have seen. This is by Jack Schaedler. He is a UI Developer for Ableton Live. So he read my post on Om and what I showed at Om was that, "Well, we can do. We can add undo." Real undo to do MVC which is sort of like a baseline MVC application in JavaScript. People think it's stupid. Well, whatever. That what people ... that's the baseline for them. So I did that and I said, "I can add undo to do MVC non-invasively in five lines of code." And I did and it just worked. Jack saw that and he was like, "Well, does that scale? Does that really work for something more interesting, something less trivial than to do MVC?" So we built this really awesome pixel editor. He's been working at it continuously for four or five months now. It basically presents a surface which is represented as an immutable vector which is the current frame and you can draw into it, and he gets all these properties without adding complexity to his application. Let me show that. You have seen it before. Again, this is different from how it was because, again, he's kept working on it. So this is a ClojureScript application. He uses core.async, he uses Om, he uses ClojureScript so I can, you know ...

Oops. It's stopped working. Let me refresh. Oh, no. That's not good. What's the ... Sorry about that. Okay. Okay, there we go.

So on the right over there, you should see that it's adding application states so I can basically do this all day. This is almost unlimited undo. I can click individual pixels, it's not going to matter. It's not going to get any slower or less responsive. On the left, you see the preview. I can scrub over it and this is like, you know, at least 30 frames per second. I can click undo. I can click redo and all that code for doing that, the preview, undo, redo, it's 60 lines of code. So this would have been a nightmare if any of you have done any type of this stuff like this. So what's cool about this is it's not like his app is any simpler. If you look at his app, his app is complicated. It's a complicated UI, right? Complicated UIs are complicated. There's no fairy dust that ClojureScript's going to magically make your thing not complicated. What ClojureScript is going to do is it's going to eliminate the unnecessary sources of complexity, right? So if he wants this feature and it doesn't complicate his program to add it or if he wants complicated inventing, core.async doesn't magically make your complicated thing just disappear but what it's going to make is that it's going to be easier to reason about, it's going to be easier to add new source of events, new consumers of events and you won't be lost in the mire of terrible stuff.

Just to wrap that bit up, one thing I did was after I saw that I was like, "Well, I wonder how much memory his app uses?" And so what I did on the right, I took an array of 4,096 elements. I just cloned it a hundred times than I just did and I used Google Chrome Developer Tools which if you're a ClojureScript Developer, you absolutely should know how Google Chrome Dev Tools work. It's critical. But I did a heap snapshot and to save 100 frames of a 64x64 pixel image a hundred times, it takes about 1.7 megabyte to use a persistent vector, and to update a persistent vector and snapshot it 100 times takes two-tenths of a megabyte. So it's getting close to an order of magnitude less memory to do the same thing.

I don't really have much more to say. Hopefully, you guys got excited about ClojureScript but the takeaway, hopefully, is that innovate. I think we have, again, we have an extremely unique opportunity with ClojureScript that a lot of other people that are targeting browsers and again, even command line JavaScript. They don't have … there's a lot more to do. Immediately, off the top of my head, things that I would like to see following the ClojureScript community is that the model story needs work. In JavaScript, when you talk about JavaScript MVCs, you have things like Backbone, you have things like Ember, to some degree Angular and basically, you have some notion of a model on the client. You can do operations on that and it's a thing. You can reason about it. Nobody's come up with anything particularly compelling for this, Om and all the other React libraries that are [inaudible 30:00] with ClojureScript, they also don't solve this problem. So the model story needs work, someone to talk about domain information. That's richer.

One interesting thing sort of take on this was DataScript. I don't know if people have seen this which is sort of exporting some elements of the … sort of, the Datomic API into the client. It is interesting. And really, what I like about it is that it allows you to store your data in a flat way which is actually, I think, much more natural for data which means that … It also means that you can build a sensible Query API of it, doing Queries on Tries is, I don't think it's fun, but Datomic has these other cool properties that your data's flat and it's easy to Query but you can also ask for entities, and entities allow you to lift a Trie out of database. And that's great for UI because widgets often take trees. So that's pretty cool and it's definitely something worth thinking about.

Also, the React model can be further improved. Some React is perfect. There are many things, there are many benefits that we get from it and definitely, our world is much improved but there, we can go further. Address-ability is a concern that actually, where I first presented Om, Rich brought it up. And that's something that we don't really have and actually even React doesn't really have. So, in traditional UI Systems, components often have an identity and that's useful. If you want to move a component or if you want to be able to precisely reference some component that you know about and that is somewhat a lost of React Model and because React lost it, Om doesn't really do too much to recover it. I do think it's recoverable that definitely needs work. It's definitely something that people should be thinking about and this is probably further out. I mean, one of the benefits of React is that that's really Facebook's problem. Facebook is spending a lot of time in the engineering resources on making it fast, making it work, you know, fast on mobile, fast on desktop and that's great. That's work that we don't have to do. But by adapting React, it's also a bit of a compromise because the primary consumer of React is the typical JavaScript Developer and the typical JavaScript Developer are not using rich data structures and that actually is throwing a bunch of optimizations out the window. For example, even in Om and as far as I can tell, in Reagent or all these other guys, they have to convert things like styles and DOM attributes back into JavaScript objects which must be walked by React. So at the bottom, we have to walk these basic properties of the virtual DOM and that sucks. If React was more immutable-friendly, even at the bottom, you have to get the performance benefits and that we get higher up in these ClojureScript React bindings.

That's actually all I have for today. Hopefully that, again, got you excited to think about what other possibilities that are worth pursuing with ClojureScript, we're building applications. Thank you very much.