Clerk: Moldable Live Programming for Clojure

Martin Kavalar
martin@nextjournal.com
nextjournal
Berlin, Germany

Philippa Markovics
philippa@nextjournal.com
nextjournal
Berlin, Germany

Jack Rusher
jack@nextjournal.com
nextjournal
Berlin, Germany

ABSTRACT

Clerk is an open source Clojure programmer's assistant that builds upon the traditions of interactive and literate programming to provide a holistic moldable development environment. Clerk layers static analysis, incremental computation, and rich browser-based graphical presentations on top of a Clojure programmer's familiar toolkit to enhance their workflow.

CCS CONCEPTS

• **Applied computing** → **Multi / mixed media creation**.

KEYWORDS

Literate Programming,Moldable Development,

ACM Reference Format:
Martin Kavalar, Philippa Markovics, and Jack Rusher . 2018. Clerk: Moldable Live Programming for Clojure. In *Proceedings of <Programming> 2023 (PX 23).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION: LITERATE PROGRAMMING, NOTEBOOKS AND INTERACTIVE DEVELOPMENT

Knuth's *Literate Programming* emphasized the importance of focusing on human beings as consumers of computer programs. His original implementation involved authoring files that combine source code and documentation, which were then divided into two derived artifacts: source code for the computer and a typeset document in natural language to explain the program.

At the same time, other software was developed to target scientific use cases rather than program documentation. These systems, which prefigured modern computational notebooks, ranged from REPL-driven approaches like Macsyma and Mathematica to integrated WYSIWYG editors like Ron Avitzur's *Milo*, PARC's *Tioga* and *Camino Real*, and commercial software like *MathCAD*.

In contemporary data science and software engineering practice, we often see interfaces that combine these two approaches, like Jupyter and Observable. In these notebooks, a user can mix prose, code, and visualizations in a single document that provides the advantages of Knuth's Literate Programming with those of a scientific computing environment. Unfortunately, most such systems require the programmer to use a browser-based editing environment (which alienates programmers with a strong investment in their own tooling) and custom file formats (which cause problems for integration with broader software engineering practices).

Although notebooks of this kind present an improvement on the programming experience of many languages, they often feel like a step backward to experienced Lisp programmers. In Lisp environments, it is common to be able to place the cursor after a single Lisp form and evaluate it in the context of a running program, providing finer granularity of control compared to the per-cell model of most notebooks. This workflow leads to a development style that these programmers are in no hurry to lose.

> That LISP users tend to prefer structured growth rather than stepwise refinement is not an effect of the programming system, since both methods are supported. I believe, however, that it is a natural consequence of the interactive development method, since programs in early stages of growth can be executed and programs in early stages of refinement cannot.
> – Erik Sandewall

At the same time, though a number of Lisp environments have included graphical presentations of program objects, most modern tooling relies on text-based representations of evaluation output and doesn't include the ability to embed widgets for direct manipulation of program state. Additionally, problems often arise when printing structurally large results, which can cause editor performance to degrade or lead to the truncation of output, and there's limited room for customization or support for requesting more data.

In comparison, interactive programming in Smalltalk-based systems has included GUI elements since the beginning, and work to further improve programmer experience along these lines has continued in Smalltalk-based systems like Self, Pharo, Glamorous Toolkit and Newspeak, which offer completely open and customizable integrated programming environments. Glamorous Toolkit, in particular, champions the idea of using easily constructed custom tools to improve productivity and reduce time spent on code archeology, which is also a big inspiration for what we'll present here.

2 PROGRAMMING WITH CLERK

> In such a future working relationship between human problem-solver and computer 'clerk', the capability of the computer for executing mathematical processes would be used whenever it was needed.
> – Douglas Engelbart

PX 23, March 13–17, 2023, Tokyo, Japan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

We have built Clerk on top of Clojure, a functional-by-default Lisp dialect primarily hosted on the Java Virtual Machine. Several aspects of the language make it an appealing target for this project:

- being a Lisp, there is limited syntax with which to contend, and the language comes with good libraries for meta-linguistic programming
- an emphasis on pure functions and immutable data structures makes static analysis easier
- when mutable state is needed, there are idiomatic thread-safe boxes that are read and updated in a functional style

While there are some rough edges around a few particularly tricky language features, these aspects have mostly worked out in our favor.

2.1 Basic Interaction: Bring-Your-Own-Editor

Clerk combines Lisp-style interactive programming with the benefits of computational notebooks, literate programming, and moldable development, all without asking programmers to abandon their favorite tools or give up their existing software engineering practices. Its design stems partially from the difficult lessons we learned after years of unsuccessfully trying to get our *own team* to use an online browser-based notebook platform that we also developed.

When working with Clerk, a split-view is typically used with a code editor next to a browser showing Clerk's representation of the same notebook, as seen in *Clerk side-by-side with Emacs*.

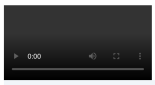


Figure: Clerk side-by-side with Emacs

As shown here, our *notebooks* are just source files containing regular Clojure code. Block comments are treated as markdown text with added support for LaTeX, data visualization, and so on, while top-level forms are treated as code cells that show the result of their evaluation. This format allows us to use Clerk in the context of production code that resides in revision control. Because files decorated with these comment blocks are legal code without Clerk loaded, they can be used in many contexts where traditional notebook-specific code cannot. This has led, among other things, to Clerk being used extensively to publish documentation for libraries that are then able to ship artifacts that have no dependency on Clerk itself.

Clerk's audience is experienced Clojure developers who are familiar with interactive development. They are able to continue programming in their accustomed style, evaluating individual forms and inspecting intermediate results, but with the added ability to `show!` a namespace/file in Clerk. A visual representation of the file is then re-computed either:

- every time the file is saved, using an an optional file watcher; or alternatively,
- via an editor hot-key that can be bound to show the current document. (The authors generally prefer the hot-key over the file watcher, as it feels more direct and gives more control over when to show something in Clerk.)

Control and configuration of Clerk primarily occurs through evaluation of Clojure forms from within the programmer's environment, rather than using outside control panels and settings. This integration with the programmer's existing tooling eases adoption and allows advanced customization of the system through code.

2.2 Fast Feedback: Caching & Incremental Computation

To keep feedback loops short, Clerk uses dependency analysis to limit recomputation to forms that haven't previously been evaluated in Clerk.

In practice this means most changes to a Clerk document are reflected instantly (within 100ms) after saving a file or hitting the keybinding to update the open document.

The caching works on the level of top-level forms. A hash is computed for each top-level form. A change to the form or one of its transitive dependencies will lead to a new hash value.

When Clerk is asked to show a notebook, it will only evaluate forms that aren't cached in one of Clerk's two caches:

- an in-memory cache stores a map of the hash of a given form to its current result. This cache is limited to the current forms of the active document.
- An on-disk-cache stores the same information but to allow the user to continue work after a restart without recomputing potentially expensive operations. Because Clojure supports lazy evaluation of potentially infinite sequences, safeguards are in place to skip caching unreasonable values.

This caching behavior can be fine-tuned (or disabled) down to the level of individual forms.

The on-disk caches use a content-addressed store where each result is stored using a filename derived from the SHA-2 hash of its contents. We use the self-describing multihash format to support future changes of the hash algorithm. Additionally, a file named after the hash of a form contains a pointer to its results filename.

This combination of immutability and indirection makes distributing the cache trivial: using last-write wins for the tiny (90 bytes) pointer files. The content-addressed result cache files are never changed and can thus be synchronized without conflict.

> While I did believe, and it has been true in practice, that the vast majority of an application could be functional, I also recognized that almost all programs would need some state. Even though the host interop would provide access to (plenty of) mutable state constructs, I didn't want state management to be the province of interop; after all, a point of Clojure was to encourage people to stop doing mutable, stateful OO. In particular I wanted a state solution that was much simpler than the inherently complex locks and mutexes approaches of the hosts for concurrency-safe state. And I wanted something that took advantage of the fact that Clojure programmers would be programming primarily with efficiently persistent immutable data.

It is idiomatic in Clojure to use boxed containers to manage mutable state. While there are several of these constructs in the language, in practice atoms are the most popular by far. An atom allows reading the current value inside it with `deref/@` and updating it's value with `swap!`.

When Clerk encounters an expression in which an atom's mutable value is being read using `deref`, it will try to compute a hash based on the value *inside* the atom at runtime, and extend the expression's static hash with it.

This extension makes Clerk's caching work naturally with idiomatic use of mutable state, and frees programmers from the need to manually opt out of caching for those expressions.

2.3 Semantic differences from regular Clojure

Clojure uses a single-pass, whole-file compilation strategy in which each evaluated form is added to the state of the running system. One positive aspect of this approach is that manually evaluating a series of forms produces the same result as loading a file containing the same forms in the same order, which is a useful property when interactively building up a program.

A practical concern with this sort of "bottom-up" programming is that the state of the system can diverge from the state of the source file, as forms that have been deleted from the source file may still be present in the running system. This can lead to a situation where newly written code depends on values that will not exist the next time the program runs, leading to surprising errors. To help avoid this, Clerk defaults to showing an error unless it can resolve all referenced definitions in the runtime to the source code.

It is our goal to match the semantics of Clojure as closely as possible but as a very dynamic language, there are limits to what Clerk's analysis can handle. Here are some of the things we currently do not support:

- Multiple definitions of the same var in a file
- Setting dynamic variables using `set!`
- Dynamically altering vars using `alter-var-root`
- Temporarily redefining vars using `with-redefs`

We have included a mechanism to override Clerk's error checking in cases where the user knows that one or more of these techniques are in use.

2.4 Presentation

Clerk uses a client/server architecture. The server runs in the JVM process that hosts the user's development environment. The client executes in a web browser running an embedded Clojure interpreter.

The process of conveying a value to the client is a *presentation*, a term taken from Common Lisp systems that support similar features . The process of presentation makes use of *viewers*, each of which is a hash map from well-known keys to quoted forms containing source code for Clojure functions that specify how the client should render data structures of a given type. When a viewer form is received on the client side, it is compiled into a function that will be then called on data later sent by the server.

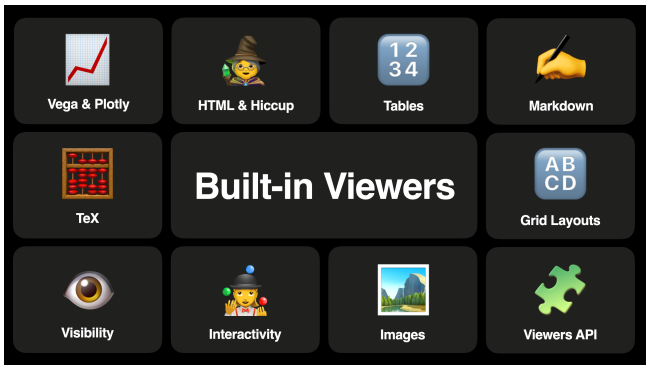
When the `present` function is called on the server side, it defaults to performing a depth-first traversal of the data structure it receives, attaching appropriate viewers at each node of the tree. The resulting structure containing both data and viewers is then sent to the client.

To avoid overloading the browser or producing uselessly large output, Clerk's built-in collection viewer carries an attribute to control the number of items initially displayed, allowing more data to be requested by the user on demand. Besides this simple limit, there's a second global *budget* per result to limit the total number of items shown in deeply nested data structures. We've found this simple system to work fairly well in practice.

One benefit of using the browser for Clerk's rendering layer is that it can produce static HTML pages for publication to the web. We could not resist the temptation to produce this document with Clerk.

It's also possible to use Clerk's presentation system in other contexts. We know of at least one case of a user leveraging Clerk's presentation system to do in-process rendering without a browser.

2.5 Built-in Viewers



Clerk comes with a set of built-in viewers for common situations. These include support for Clojure's immutable data structures, HTML (including the hiccup variant that is often used in Clojure to represent HTML and SVG), data visualization, tables, LaTeX, source code, images, and grids, as well as a fallback viewer based on Clojure's printer. The Book of Clerk gives a good overview of the available built-ins. Because Clerk's client is running in the browser, we are able to benefit from the vast JS library ecosystem. For example we're using Plotly and vega for graphing, CodeMirror for rendering code cells, and KaTeX for typesetting mathematics.

Clerk's built-in viewers try to suit themselves to typical Data Science use cases. By default, Clerk shows a code block's result as-is with some added affordances like syntax coloring and expandability of large sub-structures that are collapsed by default.

Here is an interactive example of the well-known `iris` data set, which we've added as a dependency to this notebook. Clicking the disclosure triangles will expand the data structure:

```
datasets/iris
```

```
▸[▸{:petal-length 1.4 :petal-width 0.2 :sepal-length 5.1 :sepal-width 3.5 :species "setosa"}
```

Additional affordances are automatic expansion of a nested data structure based on its shape and expanding multiple sub-structures on the same level, as demonstrated in this video:

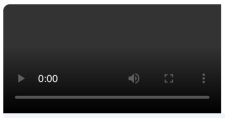


Figure: Expanding multiple sub-structures at once

Using the built-in `clerk/table` viewer, the same data structure can also be rendered as table. The table viewer is using heuristics

to infer the makeup of the table, such as column headers, from the structure of the data:

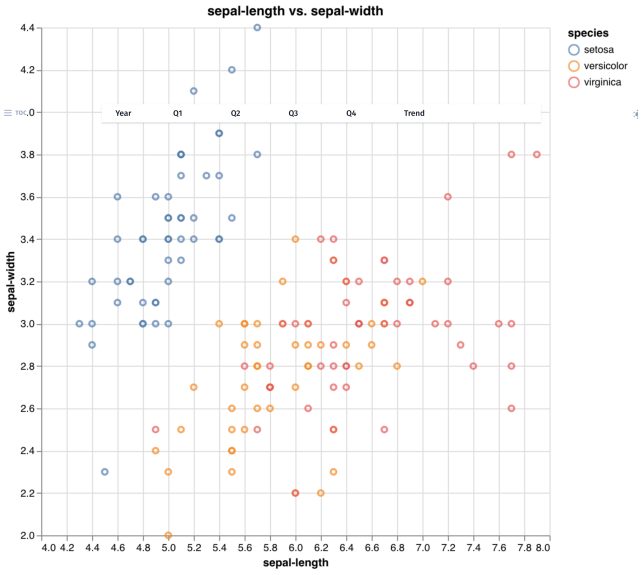
```
(clerk/table datasets/iris)
```

:sepal-length	:sepal-width	:petal-length	:petal-width	:species
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3	1.4	0.1	setosa
4.3	3	1.1	0.1	setosa
5.8	4	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

130 more...

Together with tables, plots are the most commonly used viewer for Data Science use cases. In the following figure, the same `iris` dataset, as shown in the above table example, is used to render an interactive Vega-Lite plot using the `clerk/vl` viewer:

```
(clerk/vl {:data {:values datasets/iris}
           :width 500
           :height 500
           :title "sepal-length vs. sepal-width"
           :mark {:type "point"
                  :tooltip {:field :species}}
           :encoding {:color {:field :species}
                      :x {:field :sepal-length
                          :type :quantitative
                          :scale {:zero false}}
                      :y {:field :sepal-width
                          :type :quantitative
                          :scale {:zero false}}}
           :embed/opts {:actions false}})
```



It is important to note that Clerk's viewers work in a way that encourages composition. Multiple viewers can be combined to suit a specific use case such as the following example showing a table of airline passenger numbers by year and quarter and embedding a sparkline graph into the table row for each year.

A typical Clerk workflow for this would be to first take a look at the shape of the data:

```
datasets/air-passengers
```

```
▸[▸{:month 1 :n 112 :year 1949} ▸{:month 2 :n 118 :year 1949} ▸{:month 3 :n 132 :year 1949} ▸
```

Then, a `sparkline` function is defined and tested to generate graphs (using `clerk/vl`) that will be embedded into each table row later:

```
(defn sparkline [values]
  (clerk/vl {:data {:values (map-indexed (fn [i n] {:x i :y n}) values
             :mark {:type :line :strokeWidth 1.2}
             :width 140
             :height 20
             :config {:background nil :border nil :view {:stroke "tran
             :encoding {:x {:field :x :type :ordinal :axis nil :backgr
                        :y {:field :y :type :quantitative :axis nil :b
             :embed/opts {:actions false}})))
```

```
#object▸[nextjournal.clerk.px23$sparkline 0x445255ae "nextjournal.clerk.px23$sparkline@445255
```

```
(sparkline (shuffle (range 30)))
```

Finally, the data is reduced to quarters and years, adding the sparkline graphs in a final step:

```
(clerk/table
 {:head ["Year" "Q1" "Q2" "Q3" "Q4" "Trend"]
  :rows (->> datasets/air-passengers
             (group-by :year)
             (map (fn [[year months]]
                    (let [qs (->> months (map :n) (partition 3) (map #
                      (concat [year]
```

```
                                qs
                                [(sparkline (map :n months))])))))
          (sort-by first))))
```

Year	Q1	Q2	Q3	Q4	Trend
1949	Year	Q1	Q2	Q3	Q4 Trend
1950	382	409	498	387	
1951	473	513	582	474	
1952	544	582	681	557	
1953	628	707	773	592	
1954	627	725	854	661	
1955	742	854	1023	789	
1956	878	1005	1173	883	
1957	972	1125	1336	988	
1958	1020	1146	1400	1006	
1959	1108	1288	1570	1174	
1960	1227	1468	1736	1283	

This sort of gradual refinement is a hallmark of exploratory programming, and is especially enjoyable with access to the full power of a general purpose programming language.

2.6 Moldable Viewer API

Clerk's viewers are an ordered (and thus prioritized) collection of plain Clojure hash maps. Clerk interprets the following optional keys in each viewer map:

- `:pred` is a predicate function that tests whether this viewer should be used for a given data structure
- `:transform-fn` is an optional function run on the server side to transform data before sending it to the client. It receives a map argument with the original value under a key. Additional keys carry the path, the viewer stack, and the budget (for elision)
- `:render-fn` is a quoted form that will be sent to the browser, where it will be turned into a function that will be called to display the data
- `:page-size` is a number that indicates how many items to send in each chunk during elision/pagination

Here, for example, is the `code` viewer, which shows a piece of Clojure code with idiomatic syntax highlighting:

```
{:name :code,
 :render-fn 'nextjournal.clerk.render/render-code,
 :transform-fn
 (comp
  mark-presented
  (update-val
   (fn
    [v]
    (if (string? v) v (str/trim (with-out-str (pprint/pprint v)))))))}
```

Viewers can also be explicitly selected by wrapping a value in the `clerk/with-viewer` function, which produces a presentation for that value using that viewer. Alternatively, viewers can be selected by placing a Clojure metadata declaration before a form. Because of the way Clojure handles compilation, metadata in this position is ultimately ignored in the generated code. So far as we know, this is a novel mechanism for out-of-band signaling to a specialized Clojure parser.

The process of selecting viewers happens programmatically on the server side, thus using the programmer's already existing interactive programming environment as a user interface.

2.7 Sync

To help with creating interactive tools using Clerk, it also supports bidirectional sync of state between the client and server Clojure environments. If a Clojure **`atom`** on the server is annotated with metadata indicating it is `sync`, Clerk will create a corresponding var in the client environment. Both of these atoms will be automatically instrumented with an update watcher that broadcasts a *diff* to the other side.

In addition, a server-side change will trigger a refresh of the currently active document, which will then re-calculate the minimum subset of the document that is dependent on that atom's value. This allows us to use Clerk for small local-first apps, as shown in the Regex Dictionary Example.

2.8 Tap Stream Inspector

Clerk also comes with an inspector for Clojure's tap system.

> **tap** is a shared, globally accessible system for distributing a series of informational or diagnostic values to a set of (presumably effectful) handler functions. It can be used as a better debug prn, or for facilities like logging etc.
> – Clojure 1.10 Changelog

When enabled, Clerk will attach a tap listener function and record and show the tap stream. This makes Clerk's viewer system accessible across file and namespace boundaries and independently of the caching mechanisms.

2.9 Prose-oriented Documents

The first and primary use case for Clerk was adding prose, visualizations, and interactivity to Clojure namespaces. However, when writing documents that are mainly prose, but would benefit from *some* computational elements, it is rather tedious to write everything in comment blocks. To make this easier, Clerk can also operate on markdown files with "code-fenced" source code blocks. All Clojure source blocks in such a file are evaluated and replaced in the generated document with their result.

This format is very similar to other markdown-based notebooks, like R Markdown, but specifically tailored to Clojure. We used this approach to write this paper, the source for which is located on Github.

During the review process for this paper, one of the reviewers mentioned that they would have preferred a PDF document to a website. We took this is an opportunity to test the flexibility of our system, adding a LaTeX translation layer to produce a separate printable version. It sadly lacks the interactive features of the web presentation, but we are quite pleased with the quality of the document we were able to send to press with relatively little additional work.

3 EXAMPLES OF MOLDABLE DEVELOPMENT WITH CLERK

In addition to the sorts of traditional data science use cases that one might expect from something that has "notebook" features, we intend Clerk to be a general purpose programmer's assistant that allows the rapid construction of tiny interfaces during daily work. Here are a few samples of tools and documentation created in this manner.

3.1 Augmenting table names

This example illustrates an approach we used to make working with a legacy DB2 database easier. The database's column names are made up of largely human-unreadable 8 character sequences:

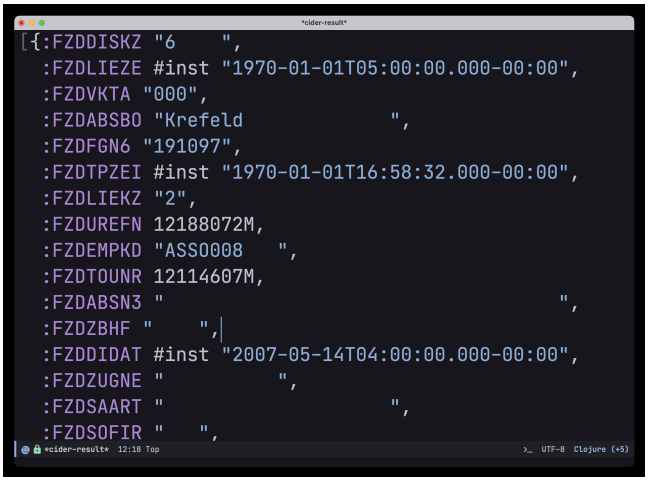


Figure 1: AS/400 Column Names

We were able to automatically translate these names using a metaschema extracted from the database. This allowed us to create a viewer that maps those eight-character names to human-readable (German-only) names (which we can then translate to English). In typical Lisp fashion, we go on to inspect a query interactively. We can use the translated names in the table, and even print them, but one quickly sees the limit of plain-text printing:

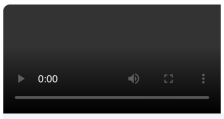


Figure: Inspecting A Query Using the REPL

With Clerk, were able to render the output as a graphical table without the limitations of plain text. Further, we can use the Viewer API to extend the table viewer's headings to show the translated metaschema names (plus showing the original eight character names in a de-emphasized way so that they aren't lost). We can go further still, showing the original German names when move the mouse over the headings:

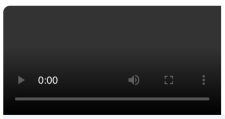


Figure: Augmented Table Headings

3.2 Rich documentation features

This example illustrates the use of Clerk to create rich documentation for `clojure2d`'s colors package. They used Clerk's Viewer API to implement custom viewers to visualize colors, gradients and color spaces, then publish that documentation on the web by generating a static website directly from the source code of the library.

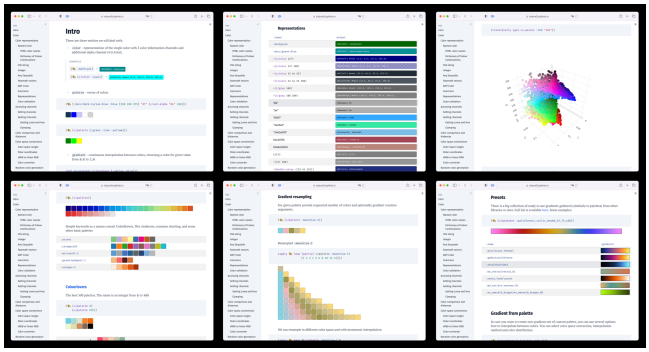


Figure 2: Custom Viewers for Clojure2d's Colors Library

3.3 Regex Dictionary

Built as a showcase for Clerk's sync feature, this example allows entering a regex into a text input and get dictionary matches as result while you type:

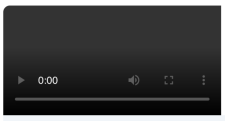


Figure: Interactive Regex Dictionary

It is built using a Clojure atom containing the text input's current value that is synced between the client and server. As you type into the input, the atom's content will be updated and synced. Consequently, printing the atom's content in your editor will show the input's current value:

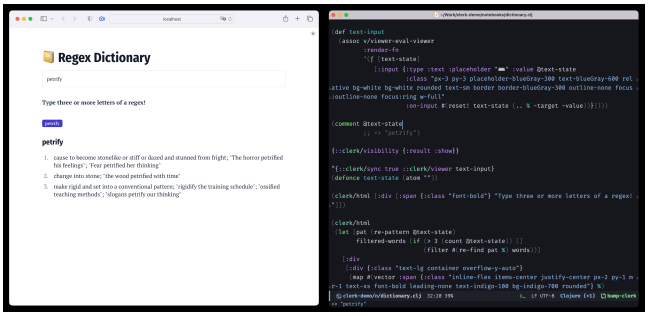


Figure 3: Printing the value of a synced Clojure atom

3.4 Lurk: Interactive Lucene-powered Log Search

Also building on Clerk's sync feature, this interactive log search uses Lucene on the JVM side to index and search a large number of log entries. In addition to using query input, logs can also be filtered by timeframe via an interactive chart. It is worth noting that this example uses a full-screen layout by opting out of Clerk's default notebook styling via Clerk's CSS customization options.

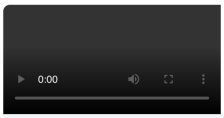


Figure: Interactive Log Search

3.5 Experience

Our experience as the developers and users of Clerk has been quite positive, but we're heavily biased. We've chosen a few quotes from Clerk's user base to give a sense of how it has been received in the community:

> [Clerk] is making the training of junior Clojure programmers a massive pleasure! [...]
> It helps us to bypass what would otherwise be a lot of distracting UI programming. Set up your env, make a namespace, hit a keybind, hey presto, your code is running in a browser.
> – Robert Stuttaford[^tweets]

> I'm using Clerk to visualize statistics properties from a simulation in a model checker [...] it's basically a wrapper over TLA+ [...]
> Amazing that Clerk just lets you focus on what really matters and nothing else!
> – Paulo Feodrippe

> I just wanted to express some gratitude for Clerk. It's been a game changer for me in terms of understanding problems and communicating that understanding to other people.
> – Jeffrey Simon

4 RELATED

Besides the systems mentioned earlier in the paper, there are a number of other contemporary related systems:

- Org mode is a major mode for Emacs supporting polyglot literate programming based on a plain text format.
- Streamlit is a Python library that eshews a custom format and enables building a web UI on regular python scripts. Its caching system memoizes functions that are tagged using Python's decorators.
- Pluto is a Julia library that uses static analysis to enable incremental computation and two-way bindings. It does come with a web-based editor. Its format are plain Julia files with comment annotations for cell ids and execution order.
- Livebook is an Elixir notebook with code editing in the browser and explicit per-cell execution. It serializes notebooks to a Markdown format.

This convergent evolution suggests to us that there is more to explore in this direction.

5 FUTURE WORK

Our goal with the development of Clerk is to *leave the toolbox open*: we want Clerk's users to be able to customize the behavior of the running system in a predictable way, often by providing functions to the system at runtime.

Clerk's viewer API is a first example of this approach, but we want to take it further by letting users:

- provide functions to control the caching e.g. to support more efficient caching of data frames
- letting the viewer API's `:pred` function opt into receiving more context like the path in the tree
- make caching more granular and support caching function invocations
- override `parse` and **`eval`** to support different syntaxes than markdown and different semantics

So far we've mainly used Clerk's caching on local machines in isolation. We plan to share a distributed cache within our dev team in order to learn about the benefits and challenges this can bring. We also want to extend Clerk to better communicate caching behavior to its users (why a value could or could not be cached, if it was cached in-memory or on-disk).

We're also actively exploring different ways of bringing an exploratory Clerk notebook to production. In exploratory work one often uses global state in Clojure atoms and vars which makes a computation tangible. When serving concurrent requests in a production setting that global state can be a source of inconsistencies.

We've been discussing ways to write changes originating from controls in Clerk's view back to source files. We also believe that for this to be a good developer experience, concurrent modifications without intermediate saving should be supported. Making a simple integration that works by overwriting source files insufficient. Because this is a significant chunk of work, and will require a different solution for each editor, we've avoided it until now. Since there certainly are many tasks for which direct manipulation can be more effective than editing text in a code editor, we're excited to explore this direction in the future.

6 CONCLUSION

We've been pleasantly surprised with how useful Clerk has been in our day-to-day work, and the adoption it has received within

the larger Clojure community suggests we are not the only ones to feel this way. We believe there are two key factors in Clerk's design that enable this:

- Enhancing regular Clojure namespaces enables incremental adoption. Neither does it need initial buy-in from the whole dev team, nor does a single member need to fully switch to working with Clerk.
- Working in concert with regular editors means it meets developers where they are and does not require a radical change of workflows.

We'd love to see folks apply these design choices to other programming ecosystems and join the pursuit to free programming from the limitations of dead text.

,