A Frontend Scheme Experiment

TODO

Additional Key Words and Phrases: Scheme, frontend, graphical user interface, web

ACM Reference Format:
. 2018. A Frontend Scheme Experiment. *Proc. ACM Meas. Anal. Comput. Syst.* 37, 4, Article 111 (August 2018), 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Scheme was created long before the World Wide Web was a thing. It is interesting to see whether Scheme can adapt to this new world. In particular, how Scheme programming language can tackle the challenges of modern frontend application development and possibly improve current practice. Through several projects, using Scheme and other programming languages, I found a way that is much simpler to think about Client-side applications and re-use existing idioms. I have put together a demo frontend application <TBA> that gives some clues about what a production application can look like.

The first section gives some history and technical background, the second section briefly describes the user interaction design of the application, the third section dive into the new components that make the forward.scm framework. The paper ends with a conclusion and further work note.

2 BACKGROUND

2.1 Toward Single-Page-Application

Historically, there are static pages that are plain html inside a directory shared over HTTP by something like Apache server. Then came the CGI era, where each request was spawning a program that was taking the request in stdin and that would output the response in stdout. This was the first incarnation of dynamic content. The main drawback of that approach is that the HTTP server had to spawn a processus for every request and to keep state around that process necessarily needed to store them somewhere in the filesystem or another kind of database. Around the same time came JavaScript that was mostly used to draw animation and slightly improved graphical components called widgets. Around 2000, XMLHTTPRequest, also known as AJAX, was made popular and allowed to query the server side without rendering a whole new page. That is the server would send an initial page with its css and javascript dependencies, then client side the JavaScript would improve the page and request only a tiny fragment of HTML or JSON to reduce the required bandwidth and speed up the interaction with the user. That is when the web 2.0 came to exist. For several reasons, more and more work was done client side. Among the reasons for that was to reduce the workload on the server, improve usability or simply to allow richer interactions. Backend side, things like FCGI came to exist and most common languages gained HTTP servers which allowed for, more or less, forward HTTP requests coming from the client to the HTTP server of the

Author's address:

2476-1249/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

backend application via a battle tested HTTP server that serves as a proxy. Frontend side practices also evolved toward bigger and bigger applications, until the Single-Page-Application (SPA). SPA, usually starts from a single index.html, app.js and a fancy style.css then the JavaScript takes control of the page, even the URL bar is under the control of JavaScript, the application only does AJAX request that fetch JSON, and never reload the page from scratch (except in case of refresh). Very recently, search engines gained support for SPA, that is SPA can be indexed by search engines. That gives less and less incentive to rely on the classic request-response page-by-page HTML rendered by the backend. Because client-side applications grew more complex, so became JavaScript and the developer experience. Browsers' Virtual Machines (VM) in charge of running JavaScript became more and more powerful. That allows Gambit's JavaScript backend to be practical. Another innovation that is happening client side is called WebAssembly. Even if it is still in the works, it makes Chibi usable to build client-side applications.

2.2 Scheme in the browser

Like explained in the previous section. Client-side applications became more and more complex, but JavaScript more or less stayed the same language with some good, some bad and some things that will stay definitely ugly because of backward compatibility. That's why better VM in the browser shed some light over a bright future for Scheme. It happens that Scheme is well-suited for frontend work, among other things, SXML allows to embed XML and HTML in Scheme code, and call-with-continuation (call/cc) allows to give a sequential look to asynchronous code. In the following paragraphs, I will describe the Scheme implementations that target the browser.

2.2.1 BiwaScheme. BiwaScheme was the first Scheme I used in the browser, it allowed me to experiment and grow my Scheme skills through small projects. The main problem with BiwaScheme is that it relies on browser support of Tail-Call-Optimization. And that is doomed to never happen.

2.2.2 Gambit. Gambit supports JavaScript as a target as part of its universal backend. There are several builds possible depending on your needs and how much bandwidth you want to waste. Still, start up times are much better than Chibi. The reason I did not use Gambit is because at the time of writing it is missing the R7RS libraries I need.

2.2.3 BiwaScheme. Chibi Scheme is an interpreter that can be compiled using emscripten to WebAssembly. Most recent browsers support WebAssembly. The only drawback is that the payload is rather big and start up time noticeable. It has good support for R7RS, and it is easy to add your own. Using message passing to call JavaScript leads to some complex implementation detail but is not a show-stopper.

3 USER INTERFACE POWERED BY ROLES AND TASKS

In most graphical user interfaces (GUI), and in particular in back-office applications, like the administration interface of a newspaper, most of the time the user-experience is constructed around "objects" and "actions" that can be applied to those objects. That is, "Create-Read-Update-Delete" operations.

That systematic approach to GUI design lead to an end-user experience that is not good in several aspects:

- Onboarding of new end-users takes time, because they must think in terms of the implementation details,
- The application and the documentation are dealt with and exposed in very different ways, which makes again difficult to onboard new users,
- Even if you know what you want to do, you do not know how to do it, because the user interface is new or different from what you know. So you need to look up the documentation (if any) and translate those steps into clicks in the real user interface,
- User feedback is difficult to comprehend, you can instrument the frontend and analyze user behavior. Still, the best feedback is direct feedback, so that is why most software projects fallback to user interviews.

There is an approach that allows to decouple the user experience from the data model, while keeping the software developer productivity at peak, a systematic approach to GUI design, that I call clickport.

Clickport takes the following concerns:

- onboarding and discoverability
- gathering user feedback
- focusing on roles and tasks

At the same time, it is meant to be easy for the software engineer to create that user interface. That is, it is a framework that provides a systematic approach to learn, grow and improve a GUI.

Instead of being object-focused, or model-focused as in Model-View-Controller, the user interface is role and task focused. A role is associated with several tasks that are related and should be done by a person or group of persons. A task is a particular action that can be executed by a given user. For instance, in the case of a newspaper: there is the writer (a role) of an article that writes (task) an article. There is the copy editor (role) that must proof-read (task) the article. And there is the editor (role) that decides what articles will appear on the frontpage (task).

All those roles and tasks call for different workflows and ideally for different user-interfaces.

At least, each user has their own workflow, so the user-interface should take into account that need and make it easy to have custom dashboards and easy access to most common tasks.

To summarize the idea of clickport, one can say that we bolt together the tasks associated with a job and their documentation, index them in a search engine, and provide an input box to query for tasks in the home page. That home page can be improved with various metrics to look like a dashboard but also with a list of available action per job depending on user's permissions. The ultimate user-interface would gamify onboarding, and shadow some tasks until the user has more experience.

The search engine allows us to query for tasks. By logging user queries, it also allows developers to discover patterns and feature requests more easily than by studying mouse travels or conducting user interviews.

4 A JOURNEY THROUGH SOME FRONTEND FRAMEWORKS

More code related, there is the framework that powers the user interface. I studied and practiced three generations of frontend frameworks, namely 1) jQuery 2) Backbone 3) ReactJS, and tried to make sense of AngularJS and VueJS. My favorite approach is by far ReactJS.

4.1 Review of existing software

4.1.1 jQuery. jQuery is meant as an abstraction layer of the W3C DOM API that is meant to work around differences between several implementations. Simply said, it implements workarounds for browser incompatibilities. It also offers some sugar over the official API. Nowadays jQuery is mostly used because its syntax is more terse; see http://youmightnotneedjquery.com/.

The main workflow of jQuery is to improve on the client the HTML sent from the server in order to provide more interactive widgets. Examples of such improvements include auto-completion, WYSIWYG HTML editors, date and time picker; see more examples at https://jqueryui.com/.

Part of this workflow is the need to update the DOM in an imperative way by removing existing DOM nodes and adding new ones and most likely registering new event handlers. A consequence of that is that one cannot easily see the result of the mutation without running the code. The intent and the resulting HTML is obfuscated inside ad-hoc algorithms that try to patch the DOM. In complicated cases, you can only keep the code manageable by removing all the nodes and rebuilding the whole DOM hierarchy from scratch, possibly losing some states like the caret position inside an input box. So, before patching you would need to compute the difference between the new and old hierarchy and apply only the changes that are relevant. That pattern was made popular much later by ReactJS.

The imperative approach to DOM construction puts a strain on developer productivity because the code is not easy to read. Also jQuery is meant to be a helper, not a framework, as such it does not provide much advice on how to structure the code. That is a pain point that BackboneJS tries to tackle.

4.1.2 BackboneJS. Backbone uses jQuery and offers some structure by the way of some kind of custom class-based Object-Oriented Programming facility that is inspired from backend frameworks like Ruby-on-Rails. It promotes separation of concerns through the Model-View-Controller pattern.

That idea is not realized completely. Backbone does not force a particular flow in the application. One can easily make terrible mistakes like doing asynchronous requests to the server inside the render method, which leads to a flickering of the widgets.

Eventually, I discovered that BackboneJS was a failed attempt at structuring frontend development. The custom class system has flaws, which leads to a broken hierarchy of class and objects that are intertwined with rendering logic.

4.1.3 ReactJS. ReactJS is a library that made the patch algorithm popular, also known as virtual DOM, as response to a common pattern found in jQuery and BackboneJS applications. The idea of ReactJS is that the developer will provide ReactJS with a virtual DOM, a hierarchy of objects that mimic the real DOM nodes, and ReactJS will "make it happen efficiently". That is, it will patch the existing DOM with only the necessary operation to produce the desired result. That approach has the advantage that the expected result is clearly described in the code. Unlike with jQuery, it does not have a custom patch algorithm for every single widget or for every single state the widget has.

Simply said, ReactJS relies on the declaration of the expected result and factors the mutation of the DOM into a single function call.

4.1.4 Redux. Redux is a state manager that wants to be explicit about the flow of data inside the application. The main idea is that the data should go always in the same direction.

Unlike BackboneJS or AngularJS or VueJS, there is no such thing as two-way bindings, where the value of a model node is linked to the value of a DOM node where data flows both ways. That is, when the model is updated by another widget the value of the DOM node gets updated, when the user changes the DOM node the model gets updated. That two-way flow of data can lead to code that is difficult to reason about, especially when you consider that the model can have hooks that are supposed to validate or trigger other side-effects in a cascade. That is, the flow of data is complex.

Still, there are several aspects where Redux falls short in terms of developer experience:

- Asynchronous calls to the backend look like an afterthought or plugin, which seems odd since standalone frontend applications are very rare. Calling the backend is the common use-case and should have support for it built-in.
- It is not clear what the "action" indirection through an enumeration does. One can read it documents the "actions" that an user can do, but so does a function.
- It is also blurry why ReduxJS recommends the use of the so-called reducers. Especially, since it is not clear how to achieve a fractal architecture: composing components (or if you prefer widgets) is not easy.

Those are the premises that lead to the construction of forward.scm.

5 FORWARD.SCM

forward.scm wants to optimize for the common case while offering a canvas of scalable patterns for single-page applications, which possibly include the distribution and re-use of components. Its design is simple enough that it allows us to fork it and adapt it to particular cases.

Here is a basic example of an application that counts clicks:

```
(define (init) 0)

(define (on-click model event)
  (+ model 1))

(define (render model)
  `(div (@ (on ((click ,on-click)))) ,model))

(create-app root init render)
```

5.1 A state manager made of procedures

The (create-app root init render) procedure takes an INIT procedure that returns the initial model for the application that is possibly immutable, and RENDER will return the SXML representation of the new DOM based on the MODEL.

create-app looks like the following:

```
(define (create-app root init view)
  (define model (init))

  (define (make-controller proc)
    (lambda args
      (set! model (apply proc model args))
      (render!)))

  (define (render! model)
    (patch! node
            make-controller
            (view model)))

  (render!))
```

The procedure create-app defines the single direction data flow of the application.

The flow of the data happens as follow:

- The initial model is returned by user provided procedure called INIT
- The user provided procedure called VIEW will be called with that model. It must return a SXML representation of desired HTML, with a special attribute called ‘on‘ that maps DOM events with event handler procedures, also called controllers. Mind the fact that view has no side effect, it only depends on its only argument model.
- At this point patch! will apply the new DOM state and register the Scheme procedures as event handlers

- When an event is fired, and an event handler is registered for it, the browser will call the associated event handler (controller). That control procedure will take the model and the DOM event as arguments and must produce a new model. Possibly doing remote procedure calls to the server.
- At this point, there is a new model, the procedure VIEW must be called with it, the process continues from the 2 with the new model.

One can add support for:

- remote procedure calls (including WebSockets),
- URL router to help with single-page applications,
- adopt a strictly functional approach like elm does,
- interact with another JavaScript application.

Most, if not all, the difficult work is done in patch!. As of the time of writing that is done with the help of ReactJS. The procedure (patch! node make-controller sxml) (which is not exposed to the user) takes a DOM node called NODE and a representation of the desired new DOM in SXML. It transforms the DOM appropriately, among other things registering new event handlers and unregistering unused event handlers.

5.2 The model

ReduxJS recommends the use of a flat data structure for the model (called store in ReduxJS). An immutable nested data structure is difficult to change. One approach is to rely on cursors. Another way is to use a persistent red-black tree. That way, there is a cheap way to implement time-travel, while re-using existing code and concepts used in the backend. At the time of writing, mostly because it allows to share the most code with the backend, a SRFI-167 interface is layered on top of the red-black tree implementation from (scheme mapping). There are two drawbacks to that approach:

- SRFI-167 requires serializing Scheme built in datatypes into byte vectors, which leads to some overhead.
- To implement an undo feature over a subset of the model it must be expressed in terms of SRFI-167 like vnstore does.

The four advantages of re-using SRFI-167, and possibly SRFI-168, are:

- One can use similar code in the backend and in the frontend, making the whole application easier to understand because concepts and code are shared between the two sides of the network,
- Scheme mapping is flat, SRFI-167 is flat but it allows thanks to its ordered nature to build higher level abstractions (full-text search, geographical indices, leaderboards, graphs…),
- It is easy to store the current state of the application in the browser's localStorage,
- You do not need cursors.

6 CONCLUSION

Working through existing JavaScript frameworks is far from a joy, from build systems to always moving packages ecosystem, it is difficult to follow the pace. I think approaches like BackboneJS, VueJS and AngularJS are wrong. There is too much magic in their approach and not enough guidelines. ReactJS inspired lots of libraries, and while it is not perfect on its own they are competitive alternatives like snabbdom or preact. Redux and Elm are a good source for inspiration and provide useful guidelines, they might be able to handle the general case, but in my opinion fail to deliver the code that eases software mastery.

There is a place for Scheme in the browser, whether it is its cursory look, always seeking for easy simplicity fueled by decades of experiences, Scheme existing or future implementations or the standards.

7 FURTHER WORK

The most notable work that was not really part of my investigation is hop.js. The reason is that hop.js is a shift of paradigm compared to current industrial practice. It tries to achieve what the browser ecosystem failed to deliver: "universal web apps" that is a single code base that degrades gracefully without JavaScript and allows for rich interactions. Also, more work remains to be done on the server side regarding HTTP, possibly getting inspiration from Python ASGI. More specifically, forward.scm is meant to be forked and adapted to specific situations and needs. I think that approach is better, still it would be nice to turn it into a product of some sort.