Model-View-Intent frontend development with Cycle.js

Term Paper

Tim Seckinger

Department of Computer Science - Fachhochschule Dortmund

Table of Contents

Abstract

Functional and reactive programming in imperative languages using observable stream libraries has grown to immense popularity in recent years, especially in web frontend development. In this paper, we explore how the Model-View-Intent pattern embraces this approach for modeling discrete-time events in human-computer-interaction and take a look at how the clean and simple architecture of the Cycle.js framework enables utilization of the pattern to create web applications fundamentally based on observables.

1. Motivation

As the evolution of web technologies enables single-page applications to become increasingly sophisticated, a multitude of open source frameworks emerge in the JavaScript ecosystem - community-driven, but sometimes also backed by large technology companies.

At the time of writing, Facebook's React (66,685 Github stars [GH-React]), Google's Angular (24,022 Github stars [GH-ng]) and its predecessor AngularJS (55,744 Github stars [GH-ngjs]) as well as Vue.js (53,434 Github stars [GH-Vue]) are leaders in popularity, while others, often still in early stages, attempt to introduce distinct new concepts to the market.
Notable examples of these include Aurelia (9,507 Github stars [GH-Aur]) with its dedication to Web Components, Svelte (4,751 Github stars [GH-Sve]) with its approach using the compilation step to eliminate any runtime remainder of the framework and Cycle.js (6,491 Github stars [GH-Cycle]) with its somewhat radical functional and reactive programming model.
Some older and well-established ones, such as Ember.js (17,847 Github stars [GH-Ember]) and Backbone.js (26,307 Github stars [GH-Bckbn]), seem to lose momentum [GTREND-EmberBckbn] as their ideas are drifting towards being considered obsolete.

1.1. The rise of functional programming

While functional programming is not a new paradigm and it has been prominent in academia for decades, only recently adoption of its principles has found its way into practical applications in the software industry.

WhatsApp, Inc. is known to use the functional programming language Erlang for most of its high load message processing [REE12]. The hybrid OOP and functional language Scala has grown to large popularity [OGR17] and has frequently been used for implementing backend systems at Twitter, Inc. [VEN09] and Zalando SE [KOP15] among others. Even classic imperative and object-oriented languages have commonly adopted functional programming concepts into language features and their core libraries, such as industry heavyweight Java introducing lambda expressions and incorporating the Streams API into the platform [GOE14], as well as the web's core language JavaScript supporting arrow functions as part of the ECMAScript® 2015 standard [ECM15] and earlier introducing typical collection operations including `map` and `filter` known from functional programming in ECMAScript® 5.1 [ECM11].

1.2. Current development of functional and reactive programming

Functional and reactive programming is a less well-known paradigm and much more recent trend, but there are also cases where it has been successfully used for building large-scale systems, for instance parallel API request processing at Netflix, Inc. [CH13].

In web frontend development, reactive and mostly functional and reactive programming has been getting a lot of attention over the past few years. In the React ecosystem, Redux (30,955 Github stars [GH-Redux]) is an immensely popular choice for application state management that - while not explicitly being communicated as functional and reactive - clearly takes lots of inspiration from the paradigms. Its largest competitor, MobX (9,130 Github stars [GH-MobX]) claims to "make state management simple and scalable by transparently applying functional reactive programming" [GH-MobX].

While Redux and other state management libraries are significantly less popular outside of the React ecosystem [GRE16], the frameworks usually have own distinct equivalents available, namely `@ngrx/store` for Angular and `vuex` for Vue.js. Angular itself has added the reactive programming library RxJS as one of just two dependencies of the core framework [GH-ng]. Elm is a functional programming language focused on web development with JavaScript as a compilation target. It provides syntax features suitable for reactive code, and despite remaining a niche

phenomenon for now, it has recently seen an increase in interest [GTREND-Elm].

Overall, with functional and reactive concepts sneaking into the world of JavaScript, we may very soon see not just the establishment of another fraction in the highly diverse and fragmented ecosystem of web applications, but a general paradigm change taking over the ecosystem from all corners at once.

If functional and reactive programming is the future of frontend web development - with Redux and RxJS being just early harbingers - how will we structure our code in future web applications? What will be the next MV* architecture once we are fully committed to functional and reactive programming?

After introducing basic functional reactive programming concepts and the Model-View-Intent architecture in the following sections of this paper, we will take a look at the MVI realization in the Cycle.js framework before closing with a conclusion and an outlook.

2. Functional and reactive programming in imperative languages

Functional reactive programming, commonly known as *FRP*, relies on the core concepts of *behaviors* and *events* [WH00]. The former represent values that change over time, while the latter occur in an instant and can optionally carry a value as well. Both behaviors and events can be combined by various operators to create new behaviors and events. A functional reactive program can arbitrarily apply these operators to existing behaviors and events as well as static values (that do not vary over time), generating new behaviors and emitting new events as its outputs [WH00].

But the widely-used programming languages at this time are mostly imperative, particularly in development of web frontends, where JavaScript dominates the market by a large margin. To translate the FRP paradigm to these languages and enable a programming style carrying its semantics, libraries have been created for a wide range of languages. The *ReactiveX* collection (also known as *Reactive Extensions*) provides a set of libraries implementing *observable sequences*, notably RxJava (24,877 Github stars [GH-RxJava]), RxSwift (9,517 Github stars [GH-RxSwift]) and RxJS (6,630 Github stars [GH-RxJS]), which have become popular in recent years [GTREND-Rx]. It is debatable whether its model of observable sequences is actually a

form of FRP, and ReactiveX does not claim to be strictly FRP because it only models values at discrete points in time [Rx-Intro]. Thus, the concept of behaviors is not truly represented in these implementations. Instead, observable sequences can be considered to only implement events in the original sense of FRP, not behaviors that vary continuously. What remains of the FRP fundamentals, however, is the notion of time being modeled explicitly in the program code as a form of input to the program.

Observable sequences - short *observables* - are data structures that can contain multiple values, like an *array*, but are at the same time asynchronous, like a *promise* in JavaScript or similar concepts in other languages. Also like a promise, they are based on a *push*-semantic, meaning that they can be subscribed to and will then push arriving values to subscribers rather than constantly being polled for new values.

Table 1. Imperative data structures

	1 value	**0..1 values**	**0..* values**
sync, pull	`T`	`T? / Optional<T>`	`T[] / Array<T>`
async, push	`Callback<T>`	`Promise<T>`	`Observable<T>`

The source of data for the observables typically lies outside of their system at the boundary to the imperative world. Examples are a stream of mouse clicks or other events provided by the runtime environment (such as a browser), but also incoming packets from sockets or results of I/O operations. The library will usually provide utility functions to create observables from these. Another common data source is time itself, allowing the creation of observables that emit values at a certain interval or once after a certain delay. The sink of data that is pushed through these observables, comes in the form of subscribers that represent another boundary, where often times side effects are performed that, for instance, alter the state of the user interface, send HTTP requests or perform other I/O operations.

The desired application logic can then be encoded by transforming and combining these streams of discrete-time events emitted by the data sources to other streams of discrete-time events that are consumed by the sinks in a pure, functional style, because side effects required by the fundamentally imperative environment have been extracted, segregated into inputs and outputs of the program and relocated to sources and sinks, respectively. The operators used to encode the desired logic on observables often resemble those known from the classic theory of functional reactive programming.

Figure 1. An example of a timer implemented using observables

```
const reset$ = Rx.Observable
  .fromEvent(resetButton, 'click') ①
  .startWith(null); ②
const second$ = reset$
  .switchMap(() => Rx.Observable.timer(0, 1000)); ③

const freeze$ = Rx.Observable
  .fromEvent(freezeButton, 'click') ④
  .startWith(false) ⑤
  .scan(freeze => !freeze); ⑥

const tick$ = Rx.Observable
  .combineLatest(second$, freeze$, (second, freeze) =>
(freeze ? null : second)) ⑦
  .filter(second => second !== null); ⑧

tick$.subscribe(tick => (timeDisplay.innerText = tick)); ⑨
```

① get an observable of clicks on the reset button

② pretend it was initially clicked once to start the timer

③ for each click, generate a new observable with values 0, 1, 2, … on corresponding seconds

④ get an observable of clicks on the freeze button

⑤ do not freeze initially

⑥ toggle freeze on each click

⑦ when the second changes or freeze is toggled, emit the current second (or `null` if frozen)

⑧ ignore null values, avoiding updates when frozen

⑨ change the displayed time whenever tick$ emits.

Figure 1 uses the RxJS library to implement a timer for a web page that ticks every second and can be reset by a button. An additional button can be used to toggle the state of the display between active (always displaying the current second) and frozen (displaying the last second before being frozen). Number one and four are *user inputs* to the program. Number three may not seem like an input at first, however, considering that we treat time as an explicit value in our system, the function given to `switchMap` depends the external property of time and is considered to be impure. Consequently, the function introduces a *temporal input* in addition to the existing user inputs. The `tick$` observable in number nine is the single output of the program, because it is the only point where a subscription occurs to perform necessary side effects.

Aside from the temporal input in number three, the `scan` and `combineLatest` operators in number six and seven demonstrate the influence of the temporal input particularly well, since both operators are specific to observables. *Scan* is similar to a *reduce* operation on an array, but each intermediate value produced by reduction is also emitted on the output observable, at consecutive instants in time. The state of the operation - i.e. the result of the reduction so far, in this case the state of the timer, *active* or *frozen* - is abstracted away from the application code by being encapsulated in the library that implements the operator. *CombineLatest* is similar to a *zip* operation on multiple arrays, but it is not required for both input observables to emit values to zip together, instead, the latest value provided by each observable is saved to combine them with a new value emitted by any of the observables. Again, the application does not model its *state* explicitly, but rather the *change* over time that affects its outputs. In this case, we avoid storing the current elapsed seconds value in our own code in spite of the possibility that we may need to render it if the timer gets unfrozen.

3. Introduction to Model-View-Intent

In modern web development, the classic *Model-View-Controller* pattern does not have as much significance as it traditionally had in architecture of most applications for decades. The first major JavaScript framework, *AngularJS*, has a concept of *controllers*, however, they do not work like traditional controllers as popularized by MVC. Controversial debates about the architecture of AngularJS applications have taken place, often suggesting *Model-View-ViewModel (MVVM)* as an appropriate backing concept, but ultimately, no classification could be applied in general.

AngularJS lead developer Igor Minar has declared it to be *"Model-View-Whatever" (MVW / MV*)* in an effort to end debates in favor of framework users' freedom to follow any approach they deem beneficial to their application architectures [MIN12].

In our reactive architecture, MVC cannot serve as the underlying pattern either, because the controller is a proactive component [Cy-MVI]. It manipulates the model and, in many interpretations, the view. A reactive approach needs to be oriented towards components that observe each other, establishing a unidirectional data flow contrary to the direction of observation. In *Model-View-Intent (MVI)*, we incorporate three components along the path of the data,

1. the *intent*,
2. the *model* and
3. the *view*.

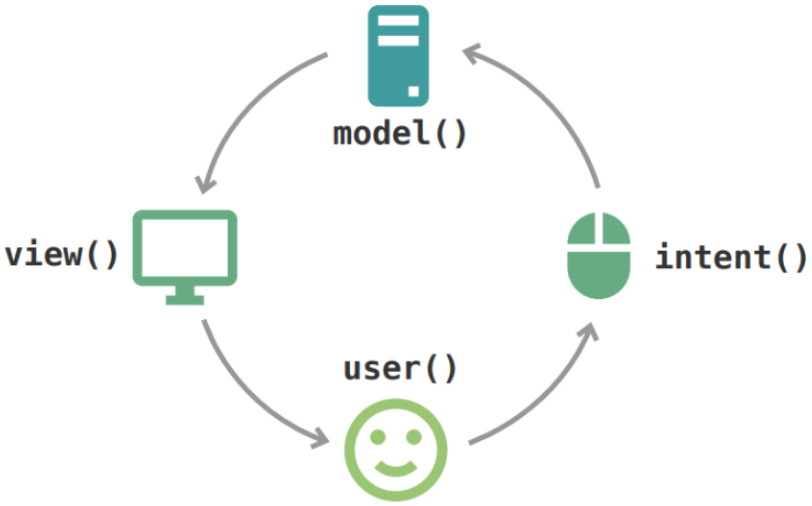

Figure 2. Visualization of MVI [STA15]

The intent observes the user's mouse clicks and keypresses or other forms of input, and infers actions to be performed from them in terms of application logic, thus interpreting the user's *intent* in giving the input.
The model holds the application state in its technical representation that can be processed by a computer. It observes the intent, adapting to incoming actions by transforming its state.
The view observes the model, rendering the application state visually and thus representing computer output in this architecture.

Figure 2 also depicts the user as a fourth component in this model. The user observes the view and incrementally builds up his mental model whenever the view changes. MVI attempts to solve the same problem that MVC tried to solve in its 1970s origins, that is translation between human and computer models, in a way that is simple and feels natural. When including the user side of the application model, the symmetry of the concept becomes obvious [STA15]. Concerning symmetry about the vertical axis, the intent transforms user output to computer input, while the view transforms computer output to user input.

More interestingly, concerning symmetry about the horizontal axis, the user holds information in his human representation, while the model holds information in its computer representation. Interaction between those occurs via the intent and view, respectively. They act like an event bus, transferring mouse clicks and screen depiction distributed over time [STA15].

Owing to this symmetry, the model can be altered to illustrate human-human-interaction in the same way it illustrates human-computer-interaction by replacing both mouse clicks and screen depiction with spoken words [STA15]. In the same way, the model can illustrate computer-computer-interaction between reactive systems via data packets sent over their event bus.

Another useful property of the MVI architecture is that every component is essentially just a referentially transparent function over streams [Cy-MVI]. Intent, model and view could - in theory - be consolidated into a single function, although the split into those three components is indispensably useful for comprehensibility of a program. At the same time, they can advantageously be split up into an arbitrary amount of functions separated by concerns, enhancing clarity and facilitating composability of MVI programs.

4. The Cycle.js framework

Cycle.js is a JavaScript framework created by André "Staltz" Medeiros, who has also been a frequent contributor to the RxJS library [Rx-Staltz]. It does not mandate use of the MVI architecture, but the concept of MVI has emerged in conjunction with Cycle.js and is beneficial to most applications that use it, with those also being the single field where MVI has been applied so far. The framework and especially its core is lightweight when compared to large frameworks such as Angular and can be viewed as more of a tool to assist in creation of MVI applications [STA15].

Cycle.js aims to provide "a functional and reactive JavaScript framework for predictable code" [GH-Cycle]. It provides its core `run` function for use with the popular RxJS functional and reactive programming library as well as *Most.js* and *xstream*, the latter of which has been developed primarily for Cycle.js. Utilities for DOM interaction, accessing other external resources such as HTTP requests and miscellaneous tooling are also available in separate packages.

In the Cycle.js architecture, side effects of any kind are encapsulated in *drivers* [Cy-Driver]. They listen to *sinks* from the application code and provide *sources* of data to the application code. Usually, data passed through the sources is based on previous sink output. For the DOM driver, the sink is a stream of DOM elements to render whenever the stream emits, and the sources are streams of events that subsequently originate from those elements. For the HTTP driver, the sink is a stream of HTTP requests to perform, and the sources are streams of the responses these requests induce. In the MVI data flow, drivers are located before the intent (sources) and after the view (sinks), interfacing with application code on one side and the external part of the system on the other - for instance the user in case of the DOM driver or another computer in case of the HTTP driver.

With all side effects extracted to drivers, only one duty remains for the application code to fulfill - mapping from provided sources to sinks according to the business logic of the desired program. Figure 3 shows an example of a simple Cycle.js application that implements a counter using the DOM driver.

Figure 3. A simple counter written using Cycle.js [Cy-Tri]

```
function main (sources) { ①
  const add$ = sources.DOM
    .select('.add') ②
    .events('click') ③
    .map(ev => 1); ④

  const count$ = add$.fold((total, change) => total +
change, 0); ⑤

  return {
    DOM: count$.map(count => ⑥
      div('.counter', [
        'Count: ' + count,
        button('.add', 'Add')
      ])
    )
  };
}

const drivers = {
  DOM: makeDOMDriver('.app') ⑦
}

run(main, drivers); ⑧
```

① The central function that contains all the logic to map sources to sinks

② Pick the element(s) of class `add`

③ Listen to `click` events

④ Emit a `1` for every click

⑤ Sum the emitted numbers, effectively increasing by 1 each time

⑥ Create a DOM element from the counter data

⑦ Declare the DOM driver and define the `app` element as the insertion point

⑧ Run the application with provided drivers

This example also visualizes the natural split of application code written for Cycle.js into

- intent (`add$`),
- model (`count$`) and
- view (return value).

The user's *intent* when clicking the button is to add 1 to the counter. The application *model* increases the stored counter state by the given change value stepwise. The *view* displays the dynamic numbers and static interaction options to the user.

The code also shows some inherent benefits of writing isolated, referentially transparent code that interfaces only with drivers according to the Cycle.js architecture. The code is automatically reusable as a component in a part of a larger Cycle.js application [Cy-Comp], because the declared driver can simply be replaced by another driver instance for other use cases. The code is also easy to test by mocking the drivers, providing mock sources and performing assertions on emitted sink values. A recent effort has been made to outsource component execution to web workers running on separate threads, communicating with the main UI thread using a modified DOM driver [Cy-SB]. This procedure is highly unusual for web frontends and hard to realize with most other framework architectures, because many architectures demand significantly more boundary points where the DOM is accessed. Additional tooling, either included in the Cycle.js codebase, or maintained by community members, is available to simplify common tasks in the Cycle.js ecosystem, including component usage and testing.

5. Conclusion

The amount of popularity the lightweight React library gained as AngularJS became fatigued can also partially be attributed to a general embracement of simplicity in tooling for web frontends. Cycle.js maxes out simplicity in comparison to component models of any other framework or library by using nothing but plain functions throughout the entire application. As the ecosystem grows and comes to fully utilize the benefits of its clean architecture, this will likely turn into a competitive advantage, especially in applications with lots of real-time aspects and user interaction.

Cycle.js also goes all the way on employment of functional and reactive programming libraries, much more so than even the newer Angular framework does. It remains to be seen if this style of programming continues expansion in frontend development. While it allows expressing the semantics of certain types of data flow in an unprecedentedly concise way, there is a large entry barrier for developers to embrace the approach because of its fundamental differences from traditional styles of programming.

The Model-View-Intent pattern may in a narrow sense be tied to the utilization of observable streams, acting as a bridge between human and computer as well as constituting arguments to and return values of the intent, model and view functions. Because of this, additional application outside of Cycle.js could be hard to find. But unidirectional data flow will likely continue to replace bidirectional data flow in frontend development, now that even Angular has dropped the latter mechanic from its predecessor AngularJS; and wherever data travels this way, the task that comes up is always to convert from user intent to application model change and, finally, view feedback.

6. Bibliography

- [GH-React] Github project page of the React library, retrieved on 2017-05-15 from https://github.com/facebook/react. Archived at http://web.archive.org/web/20170515141250/https://github.com/facebook/react.
- [GH-ng] Github project page of the Angular framework, retrieved on 2017-05-15 from https://github.com/angular/angular. Archived at http://web.archive.org/web/20170515141421/https://github.com/angular/angular.
- [GH-ngjs] Github project page of the AngularJS framework, retrieved on 2017-05-15 from https://github.com/angular/angular.js. Archived at http://web.archive.org/web/20170515141425/https://github.com/angular/angular.js.
- [GH-Vue] Github project page of the Vue.js framework, retrieved on 2017-05-15 from https://github.com/vuejs/vue. Archived at http://web.archive.org/web/20170515141530/https://github.com/vuejs/vue.
- [GH-Ember] Github project page of the Ember.js framework, retrieved on 2017-05-15 from https://github.com/emberjs/ember.js. Archived at http://web.archive.org/web/20170515142927/https://github.com/emberjs/ember.js.
- [GH-Bckbn] Github project page of the Backbone.js framework, retrieved on 2017-05-15 from https://github.com/jashkenas/backbone. Archived at http://web.archive.org/web/20170515144325/https://github.com/jashkenas/backbone.
- [GH-Aur] Github project page of the Aurelia framework, retrieved on 2017-05-15 from https://github.com/aurelia/framework. Archived at http://web.archive.org/web/20170515142932/https://github.com/aurelia/framework.
- [GH-Sve] Github project page of the Svelte framework, retrieved on 2017-05-15 from https://github.com/sveltejs/svelte. Archived at http://web.archive.org/web/20170515142936/https://github.com/sveltejs/svelte.
- [GH-Cycle] Github project page of the Cycle.js framework, retrieved on 2017-05-15 from https://github.com/cyclejs/cyclejs. Archived at http://web.archive.org/web/20170515142942/https://github.com/cyclejs/cyclejs.

- [GTREND-EmberBckbn] Google Trends diagram on ember.js vs backbone.js, retrieved on 2017-05-15 from https://trends.google.com/trends/explore?date=2010-05-16%202017-05-15&q=ember.js,backbone.js.
- [REE12] Rick Reed, "Scaling to Millions of Simultaneous Connections", presented at Erlang Factory, San Francisco Bay Area, California, USA, 2012. Slides available at http://web.archive.org/web/20170515204931/http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf.
- [OGR17] Stephen O'Grady, "The RedMonk Programming Language Rankings: January 2017", RedMonk, Portlande, Maine, USA, 2017. Report available at http://web.archive.org/web/20170515205641/http://redmonk.com/sogrady/2017/03/17/language-rankings-1-17/.
- [VEN09] Bill Venners, "Twitter on Scala - A Conversation with Steve Jenson, Alex Payne, and Robey Pointer", Artima, Inc., Mountain View, California, USA, 2009. Article available at http://web.archive.org/web/20170515210829/http://www.artima.com/scalazine/articles/twitter_on_scala.html.
- [KOP15] Alexander Kops, "From Java to Scala in Less Than Three Months", presented at Scala Days, Amsterdam, Netherlands, 2015. Slides available at http://web.archive.org/web/20170515211336/https://www.slideshare.net/ZalandoTech/zalando-tech-from-java-to-scala-in-less-than-three-months.
- [GOE14] Brian Goetz and Java Community Process contributors, "JSR-000335 Lambda Expressions for the Java™ Programming Language", Oracle Corporation, Redwood City, California, USA, 2014. Available at https://jcp.org/aboutJava/communityprocess/final/jsr335/index.html.
- [ECM15] "ECMAScript® 2015 Language Specification", Ecma International, Geneva, Switzerland, 2015. Available at https://www.ecma-international.org/ecma-262/6.0/.
- [ECM11] "ECMAScript® Language Specification (5.1 Edition)", Ecma International, Geneva, Switzerland, 2011. Available at https://www.ecma-international.org/ecma-262/5.1/.
- [CH13] Ben Christensen, Jafar Husain, "Reactive Programming in the Netflix API with RxJava", Netflix, Inc., Los Gatos, California, USA, 2013. Blog post available at http://web.archive.org/web/20170516092032/https://medium.com/netflix-techblog/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a.

- [GH-Redux] Github project page of the Redux state management library, retrieved on 2017-05-16 from https://github.com/reactjs/redux. Archived at http://web.archive.org/web/20170516092915/https://github.com/reactjs/redux.
- [GH-MobX] Github project page of the MobX state management library, retrieved on 2017-05-16 from https://github.com/mobxjs/mobx. Archived at http://web.archive.org/web/20170516093551/https://github.com/mobxjs/mobx.
- [GRE16] Sascha Greif, "State of JavaScript survey", 2016. Survey results available at http://stateofjs.com/.
- [GTREND-Elm] Google Trends diagram on Elm, retrieved on 2017-05-16 from https://trends.google.com/trends/explore?q=%2Fm%2F0ncc1sv.
- [WH00] Zhanyong Wan, Paul Hudak, "Functional reactive programming from first principles", Yale University, New Haven, Connecticut, USA, 2000. Archived at http://web.archive.org/web/20170609160109/https://pdfs.semanticscholar.org/b3b5/59104528d31f7db7fbe208377abdc4a00e15.pdf.
- [GH-RxJava] Github project page of the RxJava library, retrieved on 2017-06-10 from https://github.com/ReactiveX/RxJava. Archived at http://web.archive.org/web/20170610141854/https://github.com/ReactiveX/RxJava.
- [GH-RxSwift] Github project page of the RxSwift library, retrieved on 2017-06-10 from https://github.com/ReactiveX/RxSwift. Archived at http://web.archive.org/web/20170610141951/https://github.com/ReactiveX/RxSwift.
- [GH-RxJS] Github project page of the RxJS library, retrieved on 2017-06-10 from https://github.com/ReactiveX/rxjs. Archived at http://web.archive.org/web/20170610142030/https://github.com/ReactiveX/rxjs.
- [GTREND-Rx] Google Trends diagram on RxJava, RxSwift and RxJS, retrieved on 2017-06-10 from https://trends.google.com/trends/explore?date=2013-06-10%202017-06-10&q=RxJava,RxSwift,RxJS.
- [Rx-Intro] Introduction page to the ReactiveX collection of libraries, retrieved on 2017-06-11 from http://reactivex.io/intro.html. Archived at http://web.archive.org/web/20170611122140/http://reactivex.io/intro.html.

- [MIN12] Igor Minar, blog post on "MVC vs MVVM vs MVP", 2012, retrieved on 2017-06-27 from https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV. Archived at http://web.archive.org/web/20170627165705/https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV.
- [Cy-MVI] Cycle.js documentation page on Model-View-Intent, retrieved on 2017-06-27 from https://cycle.js.org/model-view-intent.html. Source code archived at https://github.com/cyclejs/cyclejs/blob/04a6369f62ac8b13f23b3985ead029ea362ee5bf/docs/content/documentation/model-view-intent.md.
- [STA15] André "Staltz" Medeiros, "What if the user was a function?", presented at JSConf, Budapest, Hungary, 2015. Slides available at https://speakerdeck.com/staltz/what-if-the-user-was-a-function.
- [Rx-Staltz] Github contributors graph of the RxJS library, retrieved on 2017-07-03 from https://github.com/ReactiveX/rxjs/graphs/contributors.
- [Cy-Driver] Cycle.js documentation page on drivers, retrieved on 2017-07-03 from https://cycle.js.org/drivers.html. Source code archived at https://github.com/cyclejs/cyclejs/blob/04a6369f62ac8b13f23b3985ead029ea362ee5bf/docs/content/documentation/drivers.md.
- [Cy-Tri] Cycle.js playground, retrieved on 2017-07-03 from https://cyclejs.github.io/tricycle/. Source code archived at https://github.com/cyclejs/tricycle/tree/4c0f9a11644faf5aebaed55cc53698e9e9ad6fb2.
- [Cy-Comp] Cycle.js documentation page on components, retrieved on 2017-07-03 from https://cycle.js.org/components.html. Source code archived at https://github.com/cyclejs/cyclejs/blob/04a6369f62ac8b13f23b3985ead029ea362ee5bf/docs/content/documentation/components.md.
- [Cy-SB] Github project page of the Cycle Sandbox library, retrieved on 2017-07-03 from https://github.com/aronallen/cycle-sandbox. Archived at http://web.archive.org/web/20170703151949/https://github.com/aronallen/cycle-sandbox.