Table of Contents

Derived Values, Flowing

> This, milord, is my family's axe. We have owned it for almost nine hundred years, see. Of course, sometimes it needed a new blade. And sometimes it has required a new handle, new designs on the metalwork, a little refreshing of the ornamentation ... but is this not the nine hundred-year-old axe of my family? And because it has changed gently over time, it is still a pretty good axe, y'know. Pretty good.
>
> -- Terry Pratchett, The Fifth Elephant
>   reflecting on identity, flow and derived values

clojars [re-frame "0.10.1"] license MIT License circleci passing

Introduction

- This Repo's README
- app-db (Application State)
- First Code Walk-Through
- The API
- Mental Model Omnibus

Alternative Introductions

- purelyfunctional.tv - a detailed, well written introduction.
- Lambda Island Videos - commercial videos on many clojure topics, including re-frame and reagent.

Dominoes 2 and 3 - Event Handlers

- Infographic Overview
- Effectful Handlers
- Interceptors
- Effects
- Coeffects

Domino 4 - Subscriptions

- Infographic
- Correcting a wrong
- Flow Mechanics

Domino 5 - Reagent Tutorials

- The Basics (look at the bottom of that page)
- Lambda Island Videos. There's a 3 part series.
- purelyfunctional.tv - a written overview
- Reagent Deep Dive Series from Timothy Pratley four part series
- Props, Children & Component Lifecycle by Martin Klepsch

App Structure

- Basic App Structure
- Navigation

This Repo's README

Derived Values, Flowing

> This, milord, is my family's axe. We have owned it for almost nine hundred years, see. Of course, sometimes it needed a new blade. And sometimes it has required a new handle, new designs on the metalwork, a little refreshing of the ornamentation ... but is this not the nine hundred-year-old axe of my family? And because it has changed gently over time, it is still a pretty good axe, y'know. Pretty good.
>
> -- Terry Pratchett, The Fifth Elephant
> reflecting on identity, flow and derived values

re-frame

re-frame is a pattern for writing SPAs in ClojureScript, using Reagent.

McCoy might report "It's MVC, Jim, but not as we know it". And you would respond "McCoy, you trouble maker, why even mention an OO pattern? re-frame is a **functional framework**."

Being a functional framework, it is about data, and the functions which transform that data.

Why Should You Care?

Perhaps:

1. You want to develop an SPA in ClojureScript, and you are looking for a framework.
2. You believe Facebook did something magnificent when it created React, and you are curious about the further implications. Is the combination of `reactive programming`, `functional programming` and `immutable data` going to **completely change everything**? And, if so, what would that look like in a language that embraces those paradigms?
3. You're taking a Functional Design and Programming course at San Diego State University and you have a re-frame/reagent assignment due. You've left the reading a bit late, right?
4. You know Redux, Elm, Cycle.js or Pux and you're interested in a ClojureScript implementation. In this space, re-frame is very old, hopefully in a Gandalf kind of way. First designed in Dec 2014, it even slightly pre-dates the official Elm Architecture, although thankfully we were influenced by early-Elm concepts like `foldp` and `lift`, as well as Clojure projects like Pedestal App, Om and Hoplon. Since then, re-frame has pioneered ideas like event handler middleware, coeffect accretion, and de-duplicated signal graphs.
5. Which brings us to the most important point: **re-frame is impressively buzzword compliant**. It has reactivity, unidirectional data flow, pristinely pure functions, interceptors, coeffects, conveyor belts, statechart-friendliness (FSM) and claims an immaculate hammock conception. It also has a charming xkcd reference (soon) and a hilarious, insiders-joke T-shirt, ideal for conferences (in design). What could possibly go wrong?

It Leverages Data

You might already know that ClojureScript is a modern lisp, and that lisps are **homoiconic**. If not, you do now.

That homoiconic bit is significant. It means you program in a lisp by creating and assembling lisp data structures. Dwell on that for a moment. You are **programming in data**. The functions which later transform data, themselves start as data.

Clojure programmers place particular emphasis on the primacy of data. When they aren't re-watching Rich Hickey videos, and wishing their hair was darker and more curly, they meditate on aphorisms like **Data is the ultimate in late binding**.

I cannot stress enough what a big deal this is. It may seem like a syntax curiosity at first but, when the penny drops for you on this, it tends to be a profound moment. And once you understand the importance of this concept at the language level, you naturally want to leverage similar power at the library level.

So, it will come as no surprise, then, to know that re-frame has a data oriented design. Events are data. Effects are data. DOM is data. The functions which transform data are registered and looked up via data. Interceptors (data) are preferred over middleware (higher order functions). Etc.

Data - that's the way we roll.

It is a loop

Architecturally, re-frame implements "a perpetual loop".

To build an app, you hang pure functions on certain parts of this loop, and re-frame looks after the `conveyance of data` around the loop, into and out of the transforming functions you provide - hence a tag line of "Derived Values, Flowing".

It does Physics

Remember this diagram from school? The water cycle, right?

Two distinct stages, involving water in different phases, being acted upon by different forces: gravity working one way, evaporation/convection the other.

To understand re-frame, **imagine data flowing around that loop instead of water**.

re-frame provides the conveyance of the data around the loop - the equivalent of gravity, evaporation and convection. You design what's flowing and then you hang functions off the loop at various points to compute the data's phase changes.

Sure, right now, you're thinking "lazy sod - make a proper Computer Science-y diagram". But, no. Joe Armstrong says "don't break the laws of physics" - I'm sure you've seen the videos - and if he says to do something, you do it (unless Rich Hickey disagrees, and says to do something else). So, this diagram, apart from being a plausible analogy which might help you to understand re-frame, is **practically proof** it does physics.

It is a 6-domino cascade

Computationally, each iteration of the loop involves a six domino cascade.

One domino triggers the next, which triggers the next, et cetera, boom, boom, boom, until we are back at the beginning of the loop, and the dominoes spring to attention again, ready for the next iteration of the same cascade.

The six dominoes are:

1. Event dispatch
2. Event handling
3. Effect handling
4. Query
5. View
6. DOM

1st Domino - Event Dispatch

An `event` is sent when something happens - the user clicks a button, or a websocket receives a new message.

Without the impulse of a triggering `event`, no six domino cascade occurs. It is only because of `event`s that a re-frame app is propelled, loop iteration after loop iteration, from one state to the next.

re-frame is `event` driven.

2nd Domino - Event Handling

In response to an `event`, an application must decide what action to take. This is known as `event handling`.

Event handler functions compute side effects (known in re-frame simply as `effects`). More accurately, they compute a **description of `effects`**. This description is a data structure which says, declaratively, how the world should change (because of the event).

Much of the time, only the "application state" of the SPA itself need change, but sometimes the outside world must also be affected (localstore, cookies, databases, emails, logs, etc).

3rd Domino - Effect Handling

The descriptions of `effects` are realised (actioned).

Now, to a functional programmer, `effects` are scary in a xenomorph kind of way. Nothing messes with functional purity quite like the need for side effects. On the other hand, `effects` are marvelous because they move the app forward. Without them, an app stays stuck in one state forever, never achieving anything.

So re-frame embraces the protagonist nature of `effects` - the entire, unruly zoo of them - but it does so in a controlled and largely hidden way, and in a manner which is debuggable, auditable, mockable and pluggable.

We're Now At A Pivot Point

Domino 3 just changed the world and, very often, one particular part of it: the **application state**.

re-frame's `app state` is held in one place - think of it like you would an in-memory, central database for the app (details later).

Any changes to `app state` trigger the next part of the cascade involving dominoes 4-5-6.

There's a Formula For It

The 4-5-6 domino cascade implements the formula made famous by Facebook's ground-breaking React library:
`v = f(s)`

A view, `v`, is a function, `f`, of the app state, `s`.

Said another way, there are functions `f` that compute which DOM nodes, `v`, should be displayed to the user when the application is in a given app state, `s`.

Or, to capture the dynamics we'd say: **over time**, as `s` changes, `f` will be re-run each time to compute new `v`, forever keeping `v` up to date with the current `s`.

Or, with yet another emphasis: **over time** what is presented to the user changes in response to application state changes.

In our case, domino 3 changes `s`, the application state, and, in response, dominoes 4-5-6 are concerned with re-running `f` to compute the new `v` shown to the user.

Except, of course, there are nuances. For instance, there's no single `f` to run. There may be many functions which collectively build the overall DOM, and only part of `s` may change at any one time, so only part of the `v` (DOM) need be re-computed and updated. And some parts of `v` might not be showing right now.

Domino 4 - Query

Domino 4 is about extracting data from "app state", and providing it in the right format for view functions (which are Domino 5).

Domino 4 is a novel and efficient de-duplicated signal graph which runs query functions on the app state, `s`, efficiently computing reactive, multi-layered, "materialised views" of `s`.

(Relax about any unfamiliar terminology, you'll soon see how simple the code actually is)

Domino 5 - View

Domino 5 is one or more **view functions** (aka Reagent components) that compute the UI DOM that should be displayed to the user.

To render the right UI, they need to source application state, which is delivered reactively via the queries of Domino 4. They compute hiccup-formatted data, which is a description of the DOM required.

Domino 6 - DOM

You don't write Domino 6 - it is handled for you by Reagent/React. I mention it here for completeness and to fully close the loop.

This is the step in which the hiccup-formatted "descriptions of required DOM", returned by the view functions of Domino 5, are made real. The browser DOM nodes are mutated.

Managing mutation

The two sub-cascades 1-2-3 and 4-5-6 have a similar structure.

In each, it is the second to last domino which computes "data descriptions" of mutations required, and it is the last domino which does the dirty work and realises these descriptions.

In both cases, you don't need to worry yourself about this dirty work. re-frame looks after those dominoes.

A Cascade Of Simple Functions

You'll (mostly) be writing pure functions which can be described, understood and tested independently. They take data, transform it and return new data.

The loop itself is mechanical and predictable in operation. So, there's a regularity to how a re-frame app goes about its business, which leads, in turn, to an ease in reasoning and debugging.

The Dominoes Again - With Code Fragments

So that was the view of re-frame from 60,000 feet. We'll now shift down to 30,000 feet and look again at each domino, but this time with code fragments.

Imagine: we're working on a SPA which displays a list of items. You have just clicked the "delete" button next to the 3rd item in the list.

In response, what happens within this imaginary re-frame app? Here's a sketch of the six domino cascade:

> Don't expect to completely grok the terse code presented below. We're still at 30,000 feet. Details later.

Code For Domino 1

The delete button for that 3rd item will be rendered by a ViewFunction which looks like this:

```
(defn delete-button
  [item-id]
  [:div.garbage-bin
     :on-click #(re-frame.core/dispatch [:delete-item item-id])])
```

That `on-click` handler uses re-frame's `dispatch` to emit an `event`.

A re-frame `event` is a vector and, in this case, it has 2 elements: `[:delete-item 2486]` (where `2486` is the made-up id for that 3rd item).

The first element of an event vector, `:delete-item`, is the kind of event. The rest is optional, useful data about the `event`.

Events express intent in a domain specific way. They are the language of your re-frame system.

Code For Domino 2

An `event handler` (function), called say `h`, is called to compute the `effect` of the event `[:delete-item 2486]`.

On app startup, `re-frame.core/reg-event-fx` would have been used to register this `h` as the handler for `:delete-item` events, like this:

```
(re-frame.core/reg-event-fx   ;; a part of the re-frame API
  :delete-item                ;; the kind of event
  h)                         ;; the handler function for this kind of event
```

`h` is written to take two arguments:

1. a `coeffects` map which contains the current state of the world (including app state)
2. the `event` to handle

It is the job of `h` to compute how the world should be changed by the event, and it returns a map of `effects` - a description of those changes.

Here's a sketch (we are at 30,000 feet):

```
(defn h                               ;; choose a better name like delete-item
 [coeffects event]                    ;; args:  db from coeffect, event
 (let [item-id (second event)         ;; extract id from event vector
       db      (:db coeffects)]       ;; extract the current application state
   {:db  (dissoc-in db [:items item-id])})) ;; effect is change app state
```

re-frame has ways (described in later tutorials) to inject necessary aspects of the world into that first `coeffects` argument (map). Different event handlers need different "things" to get their job done. But current "application state" is one aspect of the world which is invariably needed, and it is available by default in the `:db` key.

BTW, here is a more idiomatic rewrite of `h` which uses "destructuring" of the args:

```
(defn h
 [{:keys [db]} [_ item-id]]    ;; <--- new: obtain db and id directly
 {:db  (dissoc-in db [:items item-id])}) ;; same as before
```

Code For Domino 3

An `effect handler` (function) actions the `effects` returned by `h`.

Here's what `h` returned:

```
{:db  (dissoc-in db [:items 2486])}   ;; db is a map of some structure
```

Each key of the map identifies one kind of `effect`, and the value for that key supplies further details. The map returned by `h` only has one key, so there's only one effect.

A key of `:db` means to update the app state with the key's value.

This update of "app state" is a mutative step, facilitated by re-frame which has a built-in `effects handler` for the `:db` effect.

Why the name `:db`? Well, re-frame sees "app state" as something of an in-memory database. More on this in a following tutorial.

Just to be clear, if `h` had returned:

```
{:wear  {:pants "velour flares"  :belt false}
 :tweet "Okay, yes, I am Satoshi. #coverblown"}
```

Then, the two effects handlers registered for `:wear` and `:tweet` would be called to action those two effects. And, no, re-frame does not supply standard effect handlers for either, so you would have had to have written them yourself (see how in a later tutorial).

Code For Domino 4

We now start the `v = f(s)` part of the flow.

The application state `s` has just changed (via Domino 3) and now boom, boom go Dominoes 4, 5, and 6, at the end of which we have a new view, `v`, being shown to the user.

In this domino 4, a query (function) over this app state is automatically called. This query function "extracts" data from application state, and then computes "a materialised view" of the application state - producing data which is useful to the view functions of domino, 5.

Now, in this particular case, the query function is pretty trivial. Because the items are stored in app state, there's not a lot to compute and, instead, it acts strictly like an extractor or accessor, just plucking the list of items out of application state:

```
(defn query-fn
  [db v]         ;; db is current app state, v the query vector
  (:items db))   ;; not much of a materialised view
```

On program startup, such a `query-fn` must be associated with a `query-id`, (so it can be used via `subscribe` in domino 5) using `re-frame.core/reg-sub`, like this:

```
(re-frame.core/reg-sub  ;; part of the re-frame API
   :query-items         ;; query id
   query-fn)            ;; query fn
```

Which says "if, in domino 5, you see a `(subscribe [:query-items])`, then use `query-fn` to compute it".

Code For Domino 5

Because the query function for `:query-items` just re-computed a new value, any view (function) which uses a `(subscribe [:query-items])` is called automatically (reactively) to re-compute new DOM.

View functions compute a data structure, in hiccup format, describing the DOM nodes required. In this "items" case, the view functions will *not* be generating hiccup for the just-deleted item obviously but, other than this, the hiccup computed "this time" will be the same as "last time".

```
(defn items-view
  []
  (let [items  (subscribe [:query-items])]  ;; source items from app state
    [:div (map item-render @items)]))   ;; assume item-render already written
```

Notice how `items` is "sourced" from "app state" via `re-frame.core/subscribe`. It is called with a vector argument, and the first element of that vector is a query-id which identifies the "materialised view" required by the view.

Note: `subscribe` queries can be parameterised. So, in real world apps you might have this:
`(subscribe [:items "blue"])`

The vector identifies, first, the query, and then supplies further arguments. You could think of that as representing `select * from Items where colour="blue"`.

Except there's no SQL available and you would be the one to implement the more sophisticated `query-fn` capable of handling the "where" argument. More in later tutorials.

Code For Domino 6

The hiccup returned by the view function is made into real browser DOM by Reagent/React. No code from you required. Just happens.

The DOM computed "this time" will be the same as "last time", **except** for the absence of DOM for the deleted item, so the mutation will be to remove those now-missing DOM nodes from the browser.

3-4-5-6 Summary

The key point to understand about our 3-4-5-6 example is:

- a change to app state ...
- triggers query functions to rerun ...
- which triggers view functions to rerun
- which causes modified browser DOM

Boom, boom, boom go the dominoes. It is a reactive data flow.

Aaaaand we're done

At this point, the re-frame app returns to a quiescent state, waiting for the next event.

So, your job is ...

When building a re-frame app, you:

- design your app's information model (data and schema layer)
- write and register event handler functions (control and transition layer) (domino 2)
- (once in a blue moon) write and register effect and coeffect handler functions (domino 3) which do the mutative dirty work of which we dare not speak.
- write and register query functions which implement nodes in a signal graph (query layer) (domino 4)
- write Reagent view functions (view layer) (domino 5)

re-frame is mature and proven in the large

re-frame was released in early 2015, and has since been successfully used by quite a few companies and individuals to build complex apps, many running beyond 40K lines of ClojureScript.

Scale changes everything. Frameworks are just pesky overhead at small scale - measure them instead by how they help you tame the complexity of bigger apps, and in this regard re-frame has worked out well. Some have been effusive in their praise.

Having said that, re-frame remains a work in progress and it falls short in a couple of ways - for example it doesn't work as well as we'd like with devcards, because it is a framework, rather than a library. We're still puzzling over some aspects and tweaking as we go. All designs represent a point in the possible design space, with pros and cons.

And, yes, re-frame is fast, straight out of the box. And, yes, it has a good testing story (unit and behavioural). And, yes, it works with figwheel to create a powerful hot-loading development story. And, yes, it has fun specialist tooling, and a community, and useful 3rd party libraries.

Where Do I Go Next?

At this point, you know 50% of re-frame. The full docs are here.

There are two example apps to play with:
https://github.com/Day8/re-frame/tree/master/examples

Use a template to create your own project:
Client only: https://github.com/Day8/re-frame-template
Full Stack: http://www.luminusweb.net/

And please be sure to review these further resources:
https://github.com/Day8/re-frame/blob/master/docs/External-Resources.md

T-Shirt Unlocked

Good news. If you've read this far, your insiders T-shirt will be arriving soon - it will feature turtles, xkcd and something about "data all the way down". But we're still working on the hilarious caption bit. Open a repo issue with a suggestion.

Application State

> Well-formed Data at rest is as close to perfection in programming as it gets.
> All the crap that had to happen to put it there however...
>
> — Fogus (@fogus) April 11, 2014

The Big Ratom

re-frame puts all your application state into one place, which is called `app-db` .

Ideally, you will provide a spec for this data-in-the-one-place, using a powerful and leverageable schema.

Now, this advice is not the slightest bit controversial for 'real' databases, right? You'd happily put all your well-formed data into PostgreSQL.

But within a running application (in memory), there can be hesitation. If you have a background in OO, this data-in-one-place business is a really, really hard one to swallow. You've spent your life breaking systems into pieces, organised around behaviour and trying to hide state. I still wake up in a sweat some nights thinking about all that Clojure data lying around exposed and passive.

But, as Fogus reminds us, data at rest is quite perfect.

In re-frame, `app-db` is one of these:

```
(def app-db  (reagent/atom {}))    ;; a Reagent atom, containing a map
```

Although it is a `Reagent atom` (hereafter `ratom`), I'd encourage you to think of it as an in-memory database. It will contain structured data. You will need to query that data. You will perform CRUD and other transformations on it. You'll often want to transact on this database atomically, etc. So "in-memory database" seems a more useful paradigm than plain old map-in-atom.

Further Notes:

1. `app-state` would probably be a more accurate name, but I choose `app-db` instead because I wanted to convey the in-memory database notion as strongly as possible.
2. In the documentation and code, I make a distinction between `app-db` (the `ratom`) and `db` which is the (map) `value` currently stored **inside** this `ratom` . Be aware of that naming as you read code.
3. re-frame creates and manages an `app-db` for you, so you don't need to declare one yourself (see the the first FAQ if you want to inspect the value it holds).
4. `app-db` doesn't actually have to be a `ratom` containing a map. It could, for example, be a datascript database. In fact, any database which can signal you when it changes would do. We'd love! to be using datascript database - so damn cool - but we had too much data in our apps. If you were to use it, you'd have to tweak re-frame a bit and use Posh.

The Benefits Of Data-In-The-One-Place

1. Here's the big one: because there is a single source of truth, we write no code to synchronise state between many different stateful components. I cannot stress enough how significant this is. You end up writing less code and an entire class of bugs is eliminated. (This mindset is very different to OO which involves distributing state across objects, and then ensuring that state is synchronised, all the while trying to hide it, which is, when you think about it, quite crazy ... and I did it for years).

2. Because all app state is coalesced into one atom, it can be updated with a single `reset!` , which acts like a transactional commit. There is an instant in which the app goes from one state to the next, never a series of incremental steps which can leave the app in a temporarily inconsistent, intermediate state. Again, this simplicity causes a certain class of bugs or design problems to evaporate.

3. The data in `app-db` can be given a strong schema so that, at any moment, we can validate all the data in the application. **All of it!** We do this check after every single "event handler" runs (event handlers compute new state). And this enables us to catch errors early (and accurately). It increases confidence in the way that Types can increase confidence, only a good schema can potentially

provide more **leverage** than types.

4. Undo/Redo becomes straight forward to implement. It is easy to snapshot and restore one central value. Immutable data structures have a feature called `structural sharing` which means it doesn't cost much RAM to keep the last, say, 200 snapshots. All very efficient. For certain categories of applications (eg: drawing applications) this feature is borderline magic. Instead of undo/redo being hard, disruptive and error prone, it becomes trivial. **But,** many web applications are not self contained data-wise and, instead, are dominated by data sourced from an authoritative, remote database. For these applications, re-frame's `app-db` is mostly a local caching point, and being able to do undo/redo its state is meaningless because the authoritative source of data is elsewhere.
5. The ability to genuinely model control via FSMs (discussed later).
6. The ability to do time travel debugging, even in a production setting. More soon.

Create A Leveragable Schema

You need to create a spec schema for `app-db`. You want that leverage.

Of course, that means you'll have to learn spec and there's some overhead in that, so maybe, just maybe, in your initial experiments, you can get away without one. But not for long. Promise me you'll write a `spec`. Promise me. Okay, good.

Soon we'll look at the todomvc example which shows how to use a spec. (Check out `src/db.cljs` for the spec itself, and then in `src/events.cljs` for how to write code which checks `app-db` against this spec after every single event has been processed.)

Specs are potentially more leveragable than types. This is a big interesting idea which is not yet mainstream. Watch how:
https://www.youtube.com/watch?v=VNTQ-M_uSo8

Also, watch the mighty Rich Hickey (poor audio):
https://vimeo.com/195711510

How do I inspect it?

See FAQ #1

Previous: This Repo's README   Up: Index   Next: First Code Walk-Through

Initial Code Walk-through

At this point, you are about 55% of the way to understanding re-frame. You have:

- an overview of the 6 domino process from this repo's README
- an understanding of app state (from the previous tutorial)

In this tutorial, **we look at re-frame code**. By the end of it, you'll be at 70% knowledge, and ready to start coding an app.

What Code?

This repo contains an `/examples` application called "simple", which contains 70 lines of code. We'll look at every line of the file.

This app:

- displays the current time in a nice big, colourful font
- provides a single text input field, into which you can type a hex colour code, like "#CCC", used for the time display

When it is running, here's what it looks like:

To run the code yourself:

- Install Java 8
- Install leiningen (http://leiningen.org/#install)

Then:

1. `git clone https://github.com/Day8/re-frame.git`
2. `cd re-frame/examples/simple`
3. `lein do clean, figwheel`
4. wait a minute and then open http://localhost:3449/example.html

So, what's just happened? The ClojureScript code under `/src` has been compiled into `javascript` and put into `/resources/public/js/client.js` which is loaded into `/resources/public/example.html` (the HTML you just opened)

Figwheel provides for hot-loading, so you can edit the source code (under `/src`)and watch the loaded HTML change.

Namespace

Because this example is tiny, the source code is in a single namespace: https://github.com/Day8/re-frame/blob/master/examples/simple/src/simple/core.cljs

Within this namespace, we'll need access to both `reagent` and `re-frame` . So, at the top, we start like this:

```
(ns simple.core
  (:require [reagent.core :as reagent]
            [re-frame.core :as rf]))
```

Data Schema

Now, normally, I'd strongly recommended that you write a quality schema for your application state (the data stored in `app-db`). But, here, to minimise cognitive load, we'll cut that corner.

But ... we can't cut it completely. You'll still need an informal description, and here it is ... for this app `app-db` will contain a two-key map like this:

```
{:time       (js/Date.)  ;; current time for display
 :time-color "#f88"}     ;; the colour in which the time should be shown
```

re-frame itself owns/manages `app-db` (see FAQ #1), and it will supply the value within it (the two-key map described above) to your various handlers as required.

Events (domino 1)

Events are data.

re-frame uses a vector format for events. For example:

```
[:delete-item 42]
```

The first element in the vector is a keyword which identifies the `kind` of `event`. The further elements are optional, and can provide additional data associated with the event. The additional value above, `42`, is presumably the id of the item to delete.

Here are some other example events:

```
[:admit-to-being-satoshi false]
[:dressing/put-pants-on  "velour flares" {:method :left-leg-first :belt false}]
```

(For non-trivial applications, the `kind` keyword will be namespaced.)

Rule: events are pure data. No sneaky tricks like putting callback functions on the wire. You know who you are.

dispatch

To send an event, call `dispatch` with the event vector as argument:

```
   (rf/dispatch [:event-id  value1 value2])
```

In this "simple" app, a `:timer` event is dispatched every second:

```
(defn dispatch-timer-event
  []
  (let [now (js/Date.)]
    (rf/dispatch [:timer now])))  ;; <-- dispatch used

;; call the dispatching function every second
(defonce do-timer (js/setInterval dispatch-timer-event 1000))
```

This is an unusual source of events. Normally, it is an app's UI widgets which `dispatch` events (in response to user actions), or an HTTP POST's `on-success` handler, or a websocket which gets a new packet.

After dispatch

`dispatch` puts an event into a queue for processing.

So, **an event is not processed synchronously, like a function call**. The processing happens **later** - asynchronously. Very soon, but not now.

The consumer of the queue is a `router` which looks after the event's processing.

The `router`:

1. inspects the 1st element of an event vector
2. looks for the event handler (function) which is **registered** for this kind of event
3. calls that event handler with the necessary arguments

As a re-frame app developer, your job, then, is to write and register an event handler (function) for each kind of event.

Event Handlers (domino 2)

Collectively, event handlers provide the control logic in a re-frame application.

In this application, 3 kinds of event are dispatched: `:initialize` `:time-color-change` `:timer`

3 events means we'll be registering 3 event handlers.

Two ways to register

Event handler functions take two arguments `coeffects` and `event`, and they return `effects`.

Conceptually, you can think of `coeffects` as being "the current state of the world". And you can think of event handlers has computing and returning changes (effects) based on "the current state of the world" and the arriving event.

Event handlers can be registered via either `reg-event-fx` or `reg-event-db` (`-fx` vs `-db`):

- `reg-event-fx` can take multiple `coeffects` and can return multiple `effects`, while
- `reg-event-db` allows you to write simpler handlers for the common case where you want them to take only one `coeffect` - the current app state - and return one `effect` - the updated app state.

Because of its simplicity, we'll be using the latter here: `reg-event-db`.

reg-event-db

We register event handlers using re-frame's `reg-event-db`:

```
(rf/reg-event-db
  :the-event-id
  the-event-handler-fn)
```

The handler function you provide should expect two arguments:

- `db`, the current application state (the value contained in `app-db`)
- `v`, the event vector (what was given to `dispatch`)

So, your function will have a signature like this: `(fn [db v] ...)`.

Each event handler must compute and return the new state of the application, which means it returns a modified version of `db` (or an unmodified one, if there are to be no changes to the state).

:initialize

On startup, application state must be initialized. We want to put a sensible value into `app-db`, which starts out containing `{}`.

So a `(dispatch [:initialize])` will happen early in the app's life (more on this below), and we need to write an `event handler` for it.

Now this event handler is slightly unusual because not only does it not care about any event information passed in via the `event` vector, but it doesn't even care about the existing value in `db` - it just wants to plonk a completely new value:

```
(rf/reg-event-db              ;; sets up initial application state
  :initialize
  (fn [_ _]                  ;; the two parameters are not important here, so use _
    {:time (js/Date.)        ;; What it returns becomes the new application state
     :time-color "#f88"}))   ;; so the application state will initially be a map with two keys
```

This particular handler `fn` ignores the two parameters (usually called `db` and `v`) and simply returns a map literal, which becomes the application state.

For comparison, here's how we would have written this if we'd cared about the existing value of `db` :

```
(rf/reg-event-db
  :initialize
  (fn [db _]                 ;; we use db this time, so name it
    (-> db
      (assoc :time (js/Date.))
      (assoc :time-color "#f88")))
```

:timer

Earlier, we set up a timer function to `(dispatch [:timer now])` every second.

Here's how we handle it:

```
(rf/reg-event-db                 ;; usage:  (rf/dispatch [:timer a-js-Date])
  :timer
  (fn [db [_ new-time]]          ;; <-- de-structure the event vector
    (assoc db :time new-time)))  ;; compute and return the new application state
```

Notes:

1. the `event` will be like `[:timer a-time]` , so the 2nd `v` parameter destructures to extract the `a-time` value
2. the handler computes a new application state from `db` , and returns it

:time-color-change

When the user enters a new colour value (via an input text box):

```
(rf/reg-event-db
  :time-color-change            ;; usage:  (rf/dispatch [:time-color-change 34562])
  (fn [db [_ new-color-value]]
    (assoc db :time-color new-color-value)))   ;; compute and return the new application state
```

Effect Handlers (domino 3)

Domino 3 realises/puts into action the `effects` returned by event handlers.

In this "simple" application, our event handlers are implicitly returning only one effect: "update application state".

This particular `effect` is accomplished by a re-frame-supplied `effect` handler. **So, there's nothing for us to do for this domino**. We are using a standard re-frame effect handler.

And this is not unusual. You'll seldom have to write `effect` handlers, but in a later tutorial we'll show you more about how to do so when you need to.

Subscription Handlers (domino 4)

Subscription handlers, or `query` functions, take application state as an argument and run a query over it, returning something called a "materialised view" of that application state.

When the application state changes, subscription functions are re-run by re-frame, to compute new values (new materialised views).

Ultimately, the data returned by `query` functions is used in the `view` functions (Domino 5).

One subscription can source data from other subscriptions. So it is possible to create a tree of dependencies.

The Views (Domino 5) are the leaves of this tree. The tree's root is `app-db` and the intermediate nodes between the two are computations being performed by the query functions of Domino 4.

Now, the two examples below are trivial. They just extract part of the application state and return it. So, there's virtually no computation. A more interesting tree of subscriptions, and more explanation, can be found in the todomvc example.

reg-sub

`reg-sub` associates a `query id` with a function that computes that query, like this:

```
(rf/reg-sub
  :some-query-id  ;; query id (used later in subscribe)
  a-query-fn)     ;; the function which will compute the query
```

Then later, a view function (domino 5) subscribes to a query like this: `(subscribe [:some-query-id])` , and `a-query-fn` will be used to perform the query over the application state.

Each time application state changes, `a-query-fn` will be called again to compute a new materialised view (a new computation over app state) and that new value will be given to all `view` functions which are subscribed to `:some-query-id` . These `view` functions will then be called to compute the new DOM state (because the views depend on query results which have changed).

Along this reactive chain of dependencies, re-frame will ensure the necessary calls are made, at the right time.

Here's the code:

```
(rf/reg-sub
  :time
  (fn [db _]     ;; db is current app state. 2nd unused param is query vector
    (:time db))) ;; return a query computation over the application state

(rf/reg-sub
  :time-color
  (fn [db _]
    (:time-color db)))
```

Like I said, both of these queries are trivial. See todomvc.subs.clj for more interesting ones.

View Functions (domino 5)

`view` functions turn data into DOM. They are "State in, Hiccup out" and they are Reagent components.

An SPA will have lots of `view` functions, and collectively, they render the app's entire UI.

Hiccup

`Hiccup` is a data format for representing HTML.

Here's a trivial view function which returns hiccup-formatted data:

```
(defn greet
  []
  [:div "Hello viewers"])  ;; means <div>Hello viewers</div>
```

And if we call it:

```
(greet)
;; ==>  [:div "Hello viewers"]

(first (greet))
;; ==> :div
```

Yep, that's a vector with two elements: a keyword and a string.

Now, `greet` is pretty simple because it only has the "Hiccup Out" part. There's no "Data In".

Subscribing

To render the DOM representation of some part of the app state, view functions must query for that part of `app-db`, and that means using `subscribe`.

`subscribe` is always called like this:

```
   (rf/subscribe  [query-id some optional query parameters])
```

There's only one (global) `subscribe` function and it takes one argument, assumed to be a vector.

The first element in the vector (shown above as `query-id`) identifies the query, and the other elements are optional query parameters. With a traditional database a query might be:

```
select * from customers where name="blah"
```

In re-frame, that would be done as follows: `(subscribe [:customer-query "blah"])`, which would return a `ratom` holding the customer state (a value which might change over time!).

> Because subscriptions return a ratom, they must always be dereferenced to obtain the value. This is a recurring trap for newbies.

The View Functions

This view function renders the clock:

```
(defn clock
  []
  [:div.example-clock
   {:style {:color @(rf/subscribe [:time-color])}}
   (-> @(rf/subscribe [:time])
       .toTimeString
       (clojure.string/split " ")
       first)])
```

As you can see, it uses `subscribe` twice to obtain two pieces of data from `app-db`. If either change, re-frame will re-run this view function.

And this view function renders the input field:

```
(defn color-input
  []
  [:div.color-input
   "Time color: "
   [:input {:type "text"
            :value @(rf/subscribe [:time-color])        ;; subscribe
            :on-change #(rf/dispatch [:time-color-change (-> % .-target .-value)])}]])  ;; <---
```

Notice how it does BOTH a `subscribe` to obtain the current value AND a `dispatch` to say when it has changed.

It is very common for view functions to run event-dispatching functions. The user's interaction with the UI is usually the largest source of events.

And then a `view` function to bring the others together, which contains no subscriptions or dispatching of its own:

```
(defn ui
  []
  [:div
   [:h1 "Hello world, it is now"]
   [clock]
   [color-input]])
```

Note: `view` functions tend to be organized into a hierarchy, often with data flowing from parent to child via parameters. So, not every view function needs a subscription. Very often the values passed in from a parent component are sufficient.

Note: `view` functions should never directly access `app-db`. Data is only ever sourced via subscriptions.

Components Like Templates?

`view` functions are like the templates you'd find in Django, Rails, Handlebars or Mustache -- they map data to HTML -- except for two massive differences:

1. you have the full power of ClojureScript available to you (generating a Clojure data structure). The downside is that these are not "designer friendly" HTML templates.
2. these templates are reactive. When their input Signals change, they are automatically rerun, producing new DOM. Reagent adroitly shields you from the details, but the renderer of any `component` is wrapped by a `reaction`. If any of the "inputs" to that renderer change, the renderer is rerun.

Kick Starting The App

Below, `run` is called to kick off the application once the HTML page has loaded.

It has two tasks:

1. Load the initial application state
2. Load the GUI by "mounting" the root-level function in the hierarchy of `view` functions -- in our case, `ui` -- onto an existing DOM element.

```
(defn ^:export run
  []
  (rf/dispatch-sync [:initialize])     ;; puts a value into application state
  (reagent/render [ui]              ;; mount the application's ui into '<div id="app" />'
                  (js/document.getElementById "app")))
```

After `run` is called, the app passively waits for `events`. Nothing happens without an `event`.

When it comes to establishing initial application state, you'll notice the use of `dispatch-sync`, rather than `dispatch`. This is a simplifying cheat which ensures that a correct structure exists in `app-db` before any subscriptions or event handlers run.

Summary

Your job, when building an app, is to:

- design your app's information model (data and schema layer)
- write and register event handler functions (control and transition layer) (domino 2)
- (once in a blue moon) write and register effect and coeffect handler functions (domino 3) which do the mutative dirty work of which we dare not speak in a pure, immutable functional context. Most of the time, you'll be using standard, supplied ones.

- write and register query functions which implement nodes in a signal graph (query layer) (domino 4)
- write Reagent view functions (view layer) (domino 5)

Next Steps

You should now take time to carefully review the todomvc example application. On the one hand, it contains a lot of very helpful practical advice. On the other, it does use some more advanced features like `interceptors` which are covered later in the docs.

After that, you'll be ready to write your own code. Perhaps you will use a template to create your own project:
Client only: https://github.com/Day8/re-frame-template
Full Stack: http://www.luminusweb.net/

Obviously, you should also go on to read the further documentation.

Previous: app-db (Application State)   Up: Index   Next: The API

The re-frame API

Orientation:

1. The API is provided by `re-frame.core` :
   - at some point, it would be worth your time to browse it
   - to use re-frame, you'll need to `require` it, perhaps like this ... ```clj (ns my.namespace (:require [re-frame.core :as rf]))
   ... now use rf/reg-event-fx or rf/subscribe ```
2. The API is small. Writing an app, you use less than 10 API functions. Maybe just 5.
3. There's no auto-generated docs because of this problem but, as a substitute, the links below take you to the doc-strings of often-used API functions.

Doc Strings For API Functions

The core API consists of:

- dispatch or dispatch-sync.
- reg-event-db or reg-event-fx
- reg-sub and subscribe working together ```

Occasionally, you'll need to use:

- reg-fx
- reg-cofx and inject-cofx working together

And, finally, there are the builtin Interceptors:

- path
- after
- debug
- and browse the others

Built-in Effect Handlers

The following built-in effects are also a part of the API:

:dispatch-later

`dispatch` one or more events after given delays. Expects a collection of maps with two keys: `:ms` and `:dispatch`

usage:

```
{:dispatch-later [{:ms 200 :dispatch [:event-id "param"]}
                  {:ms 100 :dispatch [:also :this :in :100ms]}]}
```

Which means: in 200ms do this: `(dispatch [:event-id "param"])` and in 100ms ...

:dispatch

`dispatch` one event. Expects a single vector.

usage:

```
{:dispatch [:event-id "param"] }
```

:dispatch-n

`dispatch` more than one event. Expects a collection events.

usage:

```
{:dispatch-n (list [:do :all] [:three :of] [:these])}
```

Note: nil events are ignored which means events can be added conditionally:

```
{:dispatch-n (list (when (> 3 5) [:conditioned-out])
                   [:another-one])}
```

:deregister-event-handler

removes a previously registered event handler. Expects either a single id (typically a keyword), or a seq of ids.

usage:

```
{:deregister-event-handler :my-id)}
```

or:

```
{:deregister-event-handler [:one-id :another-id]}
```

:db

reset! app-db with a new value. Expects a map.

usage:

```
{:db  {:key1 value1 :key2 value2}}
```

Previous: First Code Walk-Through   Up: Index   Next: Mental Model Omnibus

> In a rush? You can skip this tutorial page on a first pass.
> It is quite abstract and it won't directly help you write re-frame code. On the other hand, it will considerably deepen your understanding of what re-frame is about, so remember to cycle back and read it later.
> Next page: Effectful Handlers

Mental Model Omnibus

> All models are wrong, but some are useful

The re-frame tutorials initially focus on the **domino narrative**. The goal is to efficiently explain the mechanics of re-frame, and to get you reading and writing code ASAP.

But **there are other perspectives** on re-frame which will deepen your understanding.

This tutorial is a tour of these ideas, justifications and insights.
It is a little rambling, but I'm hoping it will deliver for you at least one "Aaaah, I see" moment before the end.

> If a factory is torn down but the rationality which produced it is left standing, then that rationality will simply produce another factory. If a revolution destroys a government, but the systematic patterns of thought that produced that government are left intact, then those patterns will repeat themselves.
> -- Robert Pirsig, Zen and the Art of Motorcycle Maintenance

What is the problem?

First, we decided to build our SPA apps with ClojureScript, then we chose Reagent, then we had a problem. It was mid 2014.

For all its considerable brilliance, Reagent (+ React) delivers only the 'V' part of a traditional MVC framework.

But apps involve much more than V. We build quite complicated SPAs which can run to 50K lines of code. So, I wanted to know: where does the control logic go? How is state stored & manipulated? etc.

We read up on Pedestal App, Flux, Hoplon, Om, early Elm, etc., and re-frame is the architecture that emerged. Since then, we've tried to keep an eye on further developments like the Elm Architecture, Om.Next, BEST, Cycle.js, Redux, etc. They have taught us much although we have often made different choices.

re-frame does have parts which correspond to M, V, and C, but they aren't objects. It is sufficiently different in nature from (traditional, Smalltalk) MVC that calling it MVC would be confusing. I'd love an alternative.

Perhaps it is a RAVES framework - Reactive-Atom Views Event Subscription framework (I love the smell of acronym in the morning).

Or, if we distill to pure essence, `DDATWD` - Derived Data All The Way Down.

TODO: get acronym down to 3 chars! Get an image of stacked Turtles for `DDATWD` insider's joke, conference T-Shirt.

Guiding Philosophy

First, above all, we believe in the one true Dan Holmsand, creator of Reagent, and his divine instrument: the `ratom`. We genuflect towards Sweden once a day.

Second, we believe in ClojureScript, immutable data and the process of building a system out of pure functions.

Third, we believe in the primacy of data, for the reasons described in the main README. re-frame has a data oriented, functional architecture.

Fourth, we believe that Reactive Programming is one honking good idea. How did we ever live without it? It is a quite beautiful solution to one half of re-frame's data conveyance needs, **but** we're cautious about taking it too far - as far as, say, cycle.js. It doesn't take over everything in re-frame - it just does part of the job.

Finally, a long time ago in a galaxy far far away, I was lucky enough to program in Eiffel where I was exposed to the idea of command-query separation. The modern rendering of this idea is CQRS (see resources here). But, even today, we still see read/write `cursors` and two-way data binding being promoted as a good thing. Please, just say no. We already know where that goes. As your programs get bigger, the use of these two-way constructs will encourage control logic into all the wrong places and you'll end up with a tire-fire of an Architecture.
Sincerely, The Self-appointed President of the Cursor Skeptic's Society.

On DSLs and Machines

`Events` are cardinal to re-frame - they're a fundamental organising principle.

Each re-frame app will have a different set of `events` and your job is to design exactly the right ones for any given app you build. These `events` will model "intent" - generally the user's. They will be the "language of the system" and will provide the eloquence.

And they are data.

Imagine we created a drawing application. And then we allowed someone to use our application and, as they did, we captured, into a collection, the events caused by that user's actions (button clicks, drags, key presses, etc).

The collection of events might look like this:

```
(def collected-events
  [
     [:clear]
     [:new :triangle 1 2 3]
     [:select-object 23]
     [:rename "a better name"]
     [:delete-selection]
     ....
  ])
```

Now, as an aside, consider the following assembly instructions:

```
mov eax, ebx
sub eax, 216
mov BYTE PTR [ebx], 2
```

Assembly instructions are represented as data, right? Data which happens to be "executable" by the right machine - an x86 machine in the case above.

I'd like you to now look back at that collection of events and view it in the same way - data instructions which can be executed - by the right machine.

Wait. What machine? Well, the Event Handlers you register collectively implement the "machine" on which these instructions execute. When you register a new event handler using `reg-event-db`, it is like you are adding to the "instruction set" of the "machine".

In re-frame's README, near the top, I claimed that it had a Data Oriented Design. Typically, that claim means you "program" in a data structure of a certain format (Domain specific language), which is then "executed" by an interpreter.

Take hiccup as an example. It is a DSL for describing DOM. You program by supplying a data structure in a particular, known format (the DSL) and Reagent acts as the "interpreter" which executes that "language":

```
[:div {:font-size 12} "Hello"]  ;; a data structure
```

Back to re-frame. It requires that YOU design events which combine into a DSL for your app and, at the same time, it asks YOU to provide an interpreter for each instruction in that DSL. When your re-frame application runs, it is just executing a "program" (collection of events) dynamically created by the user's event-causing actions.

In summary:

- Events are the assembly language of your app.
- These instructions collectively form a Domain Specific Language (DSL). The language of your system.
- These instructions are data.
- One instruction after another gets executed by your functioning app.
- The Event Handlers you register collectively implement the "machine" on which this DSL executes.

On the subject of DSLs, watch James Reeves' excellent talk (video): Transparency through data

It does Event Sourcing

How did that error happen, you puzzle, shaking your head ruefully? What did the user do immediately prior? What state was the app in that this event was so problematic?

To debug, you need to know this information:

1. the state of the app immediately before the exception
2. What final `event` then caused your app to error

Well, with re-frame you need to record (have available):

1. A recent checkpoint of the application state in `app-db` (perhaps the initial state)
2. all the events `dispatch` ed since the last checkpoint, up to the point where the error occurred

Note: that's all just data. **Pure, lovely loggable data.**

If you have that data, then you can reproduce the error.

re-frame allows you to time travel, even in a production setting. To find the bug, install the "checkpoint" state into `app-db` and then "play forward" through the collection of dispatched events.

The only way the app "moves forwards" is via events. "Replaying events" moves you step by step towards the error causing problem.

This is perfect for debugging assuming, of course, you are in a position to capture a checkpoint of `app-db` , and the events since then.

Here's Martin Fowler's description of Event Sourcing.

It does a reduce

Here's an interesting way of thinking about the re-frame data flow ...

First, imagine that all the events ever dispatched in a certain running app were stored in a collection (yes, event sourcing again). So, if when the app started, the user clicked on button X the first item in this collection would be the event generated by that button, and then, if next the user moved a slider, the associated event would be the next item in the collection, and so on and so on. We'd end up with a collection of event vectors.

Second, remind yourself that the `combining function` of a `reduce` takes two arguments:

1. the current state of the reduction and
2. the next collection member to fold in

Then notice that `reg-event-db` event handlers take two arguments also:

1. `db` - the current state of `app-db`
2. `v` - the next event to fold in

Interesting. That's the same as a `combining function` in a `reduce` !!

So, now we can introduce the new mental model: at any point in time, the value in `app-db` is the result of performing a `reduce` over the entire `collection` of events dispatched in the app up until that time. The combining function for this reduce is the set of event handlers.

It is almost like `app-db` is the temporary place where this imagined `perpetual reduce` stores its on-going reduction.

Now, in the general case, this perspective breaks down a bit, because of `reg-event-fx` (has `-fx` on the end, not `-db`) which allows:

1. Event handlers to produce `effects` beyond just application state changes.
2. Event handlers to have `coeffects` (arguments) in addition to `db` and `v` .

But, even if it isn't the full picture, it is a very useful and interesting mental model. We were first exposed to this idea via Elm's early use of `foldp` (fold from the past), which was later enshrined in the Elm Architecture.

Derived Data All The Way Down

For the love of all that is good, please watch this terrific StrangeLoop presentation (40 mins). See what happens when you re-imagine a database as a stream!! Look at all the problems that evaporate. Think about that: shared mutable state (the root of all evil), re-imagined as a stream!! Blew my socks off.

If, by chance, you ever watched that video (you should!), you might then twig to the idea that `app-db` is really a derived value ... the video talks a lot about derived values. So, yes, app-db is a derived value of the `perpetual reduce` .

And yet, it acts as the authoritative source of state in the app. And yet, it isn't, it is simply a piece of derived state. And yet, it is the source. Etc.

This is an infinite loop of sorts - an infinite loop of derived data.

It does FSM

> Any sufficiently complicated GUI contains an ad hoc, informally-specified, bug-ridden, slow implementation of a hierarchical Finite State Machine
> -- me, trying too hard to impress my two twitter followers

`event handlers` collectively implement the "control" part of an application. Their logic interprets arriving events in the context of existing state, and they compute the new state of the application.

`events` act a bit like the `triggers` in a finite state machine, and the `event handlers` act like the rules which govern how the state machine moves from one logical state to the next.

In the simplest case, `app-db` will contain a single value which represents the current "logical state". For example, there might be a single `:phase` key which can have values like `:loading` , `:not-authenticated` `:waiting` , etc. Or, the "logical state" could be a function of many values in `app-db` .

Not every app has lots of logical states, but some do, and if you are implementing one of them, then formally recognising it and using a technique like State Charts will help greatly in getting a clean design and fewer bugs.

The beauty of re-frame, from a FSM point of view, is that all the state is in one place - unlike OO systems where the state is distributed (and synchronised) across many objects. So implementing your control logic as a FSM is fairly natural in re-frame, whereas it is often difficult and contrived in other kinds of architecture (in my experience).

So, members of the jury, I put it to you that:

- the first 3 dominoes implement an Event-driven finite-state machine
- the last 3 dominoes render of the FSM's current state for the user to observe

Depending on your app, this may or may not be a useful mental model, but one thing is for sure ...

Events - that's the way we roll.

Interconnections

Ask a Systems Theorist, and they'll tell you that a system has **parts** and **interconnections**.

Human brains tend to focus first on the **parts**, and then, later, maybe on **interconnections**. But we know better, right? We know interconnections are often critical to a system. "Focus on the lines between the boxes" we lecture anyone kind enough to listen (in my case, glassy-eyed family members).

In the case of re-frame, dominoes are the **parts**, so, tick, yes, we have looked at them first. Our brains are happy. But what about the **interconnections**?

If the **parts** are functions, as is the case with re-frame, what does it even mean to talk about **interconnections between functions?** To answer that question, I'll rephrase it as: how are the domino functions **composed**?

At the language level, Uncle Alonzo and Uncle John tell us how a function like `count` composes:

```
(str (count (filter odd?  [1 2 3 4 5])))
```

We know when `count` is called, and with what argument, and how the value it computes becomes the arg for a further function. We know how data "flows" into and out of the functions.

Sometimes, we'd rewrite this code as:

```
(->>  [1 2 3 4 5]
      (filter odd?)
      count
      str)
```

With this arrangement, we talk of "threading" data through functions. **It seems to help our comprehension to conceive function composition in terms of data flow**.

re-frame delivers architecture by supplying the interconnections - it threads the data - it composes the dominoes - it is the lines between the boxes.

But it doesn't have a universal method for this "composition". The technique it uses varies from one domino neighbour-pair to the next. Initially, it uses a queue/router, then a pipeline of interceptors and, finally, a Signal Graph.

Remember back in the original README? Our analogy for re-frame was the water cycle - water flowing around the loop, compelled by different kinds of forces at different times (gravity, convection, etc), going through phase changes.

With this focus on interconnections, we have been looking on the "forces" part of the loop. The transport.

Full Stack

If you like re-frame and want to take the principles full-stack, then these resources might be interesting to you:

Commander Pattern
https://www.youtube.com/watch?v=B1-gS0oEtYc

Datalog All The Way Down
https://www.youtube.com/watch?v=aI0zVzzoK_E

Reactive PostgreSQL: https://yogthos.net/posts/2016-11-05-LuminusPostgresNotifications.html

What Of This Romance?

My job is to be a relentless cheerleader for re-frame, right? The gyrations of my Pom-Poms should be tectonic, but the following quote makes me smile. It should be taught in all ComSci courses.

> We begin in admiration and end by organizing our disappointment
> -- Gaston Bachelard (French philosopher)

Of course, that only applies if you get passionate about a technology (a flaw of mine).

But, no. No! Those French Philosophers and their pessimism - ignore him!! Your love for re-frame will be deep, abiding and enriching.

Previous: The API Up: Index Next: Infographic Overview

Event Handling Infographics

Three diagrams are provided below:

- a beginner's romp
- an intermediate schematic depiction
- an advanced, full detail rendering

They should be reviewed in conjunction with the written tutorials.

Previous: Mental Model Omnibus Up: Index Next: Effectful Handlers

Effectful Handlers

This tutorial shows you how to implement pure event handlers that side-effect. Yes, a surprising claim.

Table Of Contents

Events Happen

Events "happen" when they are dispatched.

So, this makes an event happen:

```
(dispatch [:repair-ming-vase true])
```

Events are normally triggered by an external agent: the user clicks a button, or a server-pushed message arrives on a websocket.

Handling The Happening

Once dispatched, an event must be "handled" - which means it must be processed or actioned.

Events are mutative by nature. If your application is in one state before an event is processed, it will be in a different state afterwards.

And that state change is very desirable. Without the state change our application can't incorporate that button click, or the newly arrived websocket message. Without mutation, an app would just sit there, stuck.

State change is how an application "moves forward" - how it does its job. Useful!

On the other hand, control logic and state mutation tend to be the most complex and error prone part of an app.

Your Handling

To help wrangle this potential complexity, re-frame's introduction provided you with a simple programming model.

It said you should call `reg-event-db` to associate an event id, with a function to do the handling:

```
(re-frame.core/reg-event-db        ;; <-- call this to register a handler
     :set-flag                     ;; this is an event id
    (fn [db [_ new-value]]         ;; this function does the handling
       (assoc db :flag new-value)))
```

The function you register, handles events with a given `id`.

And that handler `fn` is expected to be pure. Given the value in `app-db` as the first argument, and the event (vector) as the second argument, it is expected to provide a new value for `app-db`.

Data in, a computation and data out. Pure.

90% Solution

This paradigm provides a lovely solution 90% of the time, but there are times when it isn't enough.

Here's an example from the messy 10%. To get its job done, this handler has to side effect:

```
(reg-event-db
   :my-event
   (fn [db [_ bool]]
       (dispatch [:do-something-else 3])    ;; oops, side-effect
       (assoc db :send-spam new-val)))
```

That `dispatch` queues up another event to be processed. It changes the world.

Just to be clear, this code works. The handler returns a new version of `db`, so tick, and that `dispatch` will itself be "handled" asynchronously very shortly after this handler finishes, double tick.

So, you can "get away with it". But it ain't pure.

And here's more carnage:

```
(reg-event-db
   :my-event
   (fn [db [_ a]]
       (GET "http://json.my-endpoint.com/blah"   ;; dirty great big side-effect
            {:handler       #(dispatch [:process-response %1])
             :error-handler #(dispatch [:bad-response %1])})
       (assoc db :flag true)))
```

Again, this approach will work. But that dirty great big side-effect doesn't come for free. It's like a muddy monster truck has shown up in our field of white tulips.

Bad, Why?

The moment we stop writing pure functions there are well documented consequences:

1. Cognitive load for the function's later readers goes up because they can no longer reason locally.
2. Testing becomes more difficult and involves "mocking". How do we test that the http GET above is using the right URL? "mocking" should be mocked. It is a bad omen.
3. And event replay-ability is lost.

Regarding the 3rd point above, a re-frame application proceeds step by step, like a reduce. From the README:

> at any one time, the value in app-db is the result of performing a reduce over the entire collection of events dispatched in the app up until that time. The combining function for this reduce is the set of registered event handlers.

Such a collection of events is replay-able which is a dream for debugging and testing. But only when all the handlers are pure. Handlers with side-effects (like that HTTP GET, or the `dispatch`) pollute the replay, inserting extra events into it, etc., which ruins the process.

The 2nd Kind Of Problem

And there's the other kind of purity problem:

```
(reg-event-db
   :load-localstore
   (fn [db _]
     (let [val (js->clj (.getItem js/localStorage "defaults-key"))]  ;; <-- Problem
       (assoc db :defaults val))))
```

You'll notice the event handler obtains data from LocalStore.

Although this handler has no side effect - it doesn't need to change the world - that action of obtaining data from somewhere other than its arguments, means it isn't pure.

Effects And Coeffects

When striving for pure event handlers there are two considerations:

- **Effects** - what your event handler does to the world (aka side-effects)
- **Coeffects** - the data your event handler requires from the world in order to do its computation (aka side-causes)

We'll need a solution for both.

Why Does This Happen?

It is inevitable that, say, 10% of your event handlers have effects and coeffects.

They have to implement the control logic of your re-frame app, which means dealing with the outside, mutative world of servers, databases, window.location, LocalStore, cookies, etc.

There's just no getting away from living in a mutative world, which sounds pretty ominous. Is that it? Are we doomed to impurity?

Well, luckily a small twist in the tale makes a profound difference. We will look at side-effects first. Instead of creating event handlers which *do side-effects*, we'll instead get them to *cause side-effects*.

Doing vs Causing

I proudly claim that this event handler is pure:

```
(reg-event-db
   :my-event
   (fn [db _]
      (assoc db :flag true)))
```

Takes a `db` value, computes and returns a `db` value. No coeffects or effects. Yep, that's Pure!

Yes, all true, but ... this purity is only possible because re-frame is doing the necessary side-effecting.

Wait on. What "necessary side-effecting"?

Well, application state is stored in `app-db`, right? And it is a ratom. And after each event handler runs, it must be `reset!` to the newly returned value. Notice `reset!`. That, right there, is the "necessary side effecting".

We get to live in our ascetic functional world because re-frame is looking after the "necessary side-effects" on `app-db`.

Et tu, React?

Turns out it's the same pattern with Reagent/React.

We get to write a nice pure component, like:

```
(defn say-hi
    [name]
    [:div "Hello " name])
```

and Reagent/React mutates the DOM for us. The framework is looking after the "necessary side-effects".

Pattern Structure

Pause and look back at `say-hi`. I'd like you to view it through the following lens: it is a pure function which **returns a description of the side-effects required**. It says: add a div element to the DOM.

Notice that the description is declarative. We don't tell React how to do it.

Notice also that it is data. Hiccup is just vectors and maps.

This is a big, important concept. While we can't get away from certain side-effects, we can program using pure functions which **describe side-effects, declaratively, in data** and let the backing framework look after the "doing" of them. Efficiently. Discreetly.

Let's use this pattern to solve the side-effecting event-handler problem.

Effects: The Two Step Plan

From here, two steps:

1. Work out how event handlers can declaratively describe side-effects, in data.
2. Work out how re-frame can do the "necessary side-effecting". Efficiently and discreetly.

Step 1 Of Plan

So, how would it look if event handlers returned side-effects, declaratively, in data?

Here is an impure, side effecting handler:

```
(reg-event-db
   :my-event
   (fn [db [_ a]]
       (dispatch [:do-something-else 3])    ;; <-- Eeek, side-effect
       (assoc db :flag true)))
```

Here it is re-written so as to be pure:

```
(reg-event-fx                              ;; <1>
   :my-event
   (fn [{:keys [db]} [_ a]]                ;; <2>
      {:db  (assoc db :flag true)          ;; <3>
       :dispatch [:do-something-else 3]}))
```

Notes:
<1> we're using `reg-event-fx` instead of `reg-event-db` to register (that's `-db` vs `-fx`)
<2> the first parameter is no longer just `db` . It is a map from which we are destructuring db, i.e. it is a map which contains a `:db` key.
<3> The handler is returning a data structure (map) which describes two side-effects:

- a change to application state, via the `:db` key
- a further event, via the `:dispatch` key

Above, the impure handler **did** a `dispatch` side-effect, while the pure handler **described** a `dispatch` side-effect.

Another Example

The impure way:

```
(reg-event-db
   :my-event
   (fn [db [_ a]]
       (GET "http://json.my-endpoint.com/blah"   ;; dirty great big side-effect
            {:handler       #(dispatch [:process-response %1])
             :error-handler #(dispatch [:bad-response %1])})
       (assoc db :flag true)))
```

the pure, descriptive alternative:

```
(reg-event-fx
   :my-event
   (fn [{:keys [db]} [_ a]]
       {:http {:method :get
               :url    "http://json.my-endpoint.com/blah"
               :on-success  [:process-blah-response]
               :on-fail     [:failed-blah]}
        :db   (assoc db :flag true)}))
```

Again, the old way **did** a side-effect (Booo!) and the new way **describes**, declaratively, in data, the side-effects required (Yaaa!).

More on side effects in a minute, but let's double back to coeffects.

The Coeffects

So far we've written our new style `-fx` handlers like this:

```
(reg-event-fx
   :my-event
   (fn [{:keys [db]} event]   ;; <--  destructuring to get db
       { ... }))
```

It is now time to name that first argument:

```
(reg-event-fx
   :my-event
   (fn [cofx event]       ;;  <--- thy name be cofx
       { ... }))
```

When you use the `-fx` form of registration, the first argument of your handler will be a map of coeffects which we name `cofx`.

In that map will be the complete set of "inputs" required by your function. The complete set of computational resources (data) needed to perform its computation. But how? This will be explained in an upcoming tutorial, I promise, but for the moment, take it as a magical given.

One of the keys in `cofx` will likely be `:db` and that will be the value of `app-db`.

Remember this impure handler from before:

```
(reg-event-db            ;;  a -db registration
 :load-localstore
 (fn [db _]              ;; db first argument
  (let [defaults (js->clj (.getItem js/localStorage "defaults-key"))]  ;; <--  Eeek!!
    (assoc db :defaults defaults))))
```

It was impure because it obtained an input from other than its arguments. We'd now rewrite it as a pure handler, like this:

```
(reg-event-fx             ;; notice the -fx
   :load-localstore
   (fn [cofx  _]          ;; cofx is a map containing inputs
     (let [defaults (:local-store cofx)]  ;; <--  use it here
       {:db (assoc (:db cofx) :defaults defaults)})))  ;; returns effects map
```

So, by some magic, not yet revealed, LocalStore will be queried before this handler runs and the required value from it will be placed into `cofx` under the key `:localstore` for the handler to use.

That process leaves the handler itself pure because it only sources data from arguments.

Variations On A Theme

`-db` handlers and `-fx` handlers are conceptually the same. They only differ numerically.

`-db` handlers take **one** coeffect called `db`, and they return only **one** effect (db again).

Whereas `-fx` handlers take potentially **many** coeffects (a map of them) and they return potentially **many** effects (a map of them). So, One vs Many.

Just to be clear, the following two handlers achieve the same thing:

```
(reg-event-db
   :set-flag
   (fn [db [_ new-value]]
      (assoc db :flag new-value)))
```

vs

```
(reg-event-fx
   :set-flag
   (fn [cofx [_ new-value]]
      {:db (assoc (:db cofx) :flag new-value)}))
```

Obviously the `-db` variation is simpler and you'd use it whenever you can. The `-fx` version is more flexible, so it will sometimes have its place.

Summary

90% of the time, simple `-db` handlers are the right tool to use.

But about 10% of the time, our handlers need additional inputs (coeffects) or they need to cause additional side-effects (effects). That's when you reach for `-fx` handlers.

`-fx` handlers allow us to return effects, declaratively in data.

In the next tutorial, we'll shine a light on `interceptors` which are the mechanism by which event handlers are executed. That knowledge will give us a springboard to then, as a next step, better understand coeffects and effects. We'll soon be writing our own.

Previous: Infographic Overview   Up: Index   Next: Interceptors

re-frame Interceptors

This tutorial explains re-frame Interceptors. By the end, you'll much better understand the mechanics of event handling.

As you read this, refer back to the 3rd panel of the Infographic.

Table Of Contents

Why Interceptors?

Two reasons.

First, we want simple event handlers.

Interceptors can look after "cross-cutting" concerns like undo, tracing and validation. They help us to factor out commonality, hide complexity and introduce further steps into the "Derived Data, Flowing" story promoted by re-frame.

So, you'll want to use Interceptors because they solve problems, and help you to write nice code.

Second, under the covers, Interceptors provide the mechanism by which event handlers are executed (when you `dispatch`). They are a central concept.

What Do Interceptors Do?

They wrap.

Specifically, they wrap event handlers.

Imagine your event handler is like a piece of ham. An interceptor would be like bread on either side of your ham, which makes a sandwich.

And two Interceptors, in a chain, would be like you put another pair of bread slices around the outside of the existing sandwich to make a sandwich of the sandwich. Now it is a very thick sandwich.

Interceptors wrap on both sides of a handler, layer after layer.

Wait, I know That Pattern!

Interceptors implement middleware. But differently.

Traditional middleware - often seen in web servers - creates a data processing pipeline via the nested composition of higher order functions. The result is a "stack" of functions. Data flows through this pipeline, first forwards from one end to the other, and then backwards.

Interceptors achieve the same outcome by assembling functions, as data, in a collection (a chain, rather than a stack). Data can then be iteratively pipelined, first forwards through the functions in the chain, and then backwards along the same chain.

Because the interceptor pipeline is composed via data, rather than higher order functions, it is a more flexible arrangement.

What's In The Pipeline?

Data. It flows through the pipeline being progressively transformed.

Fine. But what data?

With a web server, the middleware "stack" progressively transforms a `request` in one direction, and, then in the backwards sweep, it progressively produces a `response`.

In re-frame, the forwards sweep progressively creates the `coeffects` (inputs to the event handler), while the backwards sweep processes the `effects` (outputs from the event handler).

I'll pause while you read that sentence again. That's the key concept, right there.

Show Me

At the time when you register an event handler, you can provide a chain of interceptors too.

Using a 3-arity registration function:

```
(reg-event-db
   :some-id
   [in1 in2]       ;; <--- a chain of 2 interceptors
   (fn [db v]      ;; <-- the handler here, as before
      ....)))
```

> Each Event Handler can have its own tailored interceptor chain, provided at registration-time.

Handlers Are Interceptors Too

You might see that registration above as associating `:some-id` with two things: (1) a chain of 2 interceptors `[in1 in2]` and (2) a handler.

Except, the handler is turned into an interceptor too (we'll see how shortly).

So `:some-id` is only associated with one thing: a 3-chain of interceptors, with the handler wrapped in an interceptor, called say `h`, and put on the end of the other two: `[in1 in2 h]`.

Except, the registration function itself, `reg-event-db`, actually takes this 3-chain and inserts its own standard interceptors, called say `std1` and `std2` (which do useful things, more soon) at the front, so **ACTUALLY**, there's about 5 interceptors in the chain: `[std1 std2 in1 in2 h]`

So, ultimately, that event registration associates the event id `:some-id` with **just** a chain of interceptors. Nothing more.

Later, when a `(dispatch [:some-id ...])` happens, that 5-chain of interceptors will be "executed". And that's how an event gets handled.

Executing A Chain

The Links Of The Chain

Each interceptor has this form:

```
{:id      :something             ;; decorative only
 :before  (fn [context] ...)     ;; returns possibly modified context
 :after   (fn [context] ...)}    ;; `identity` would be a noop
```

That's essentially a map of two functions. Now imagine a vector of these maps - that's an interceptor chain.

Above we imagined an interceptor chain of `[std1 std2 in1 in2 h]`. Now we know that this is really a vector of 5 maps: `[{...} {...} {...} {...} {...}]` where each of the 5 maps have a `:before` and `:after` fn.

Sometimes, the `:before` and `:after` fns are noops (think `identity`).

To "execute" an interceptor chain:

1. create a `context` (a map, described below)
2. iterate forwards over the chain, calling the `:before` function on each interceptor
3. iterate over the chain in the opposite direction calling the `:after` function on each interceptor

Remember that the last interceptor in the chain is the handler itself (wrapped up to be the `:before`).

That's it. That's how an event gets handled.

What Is Context?

Some data called a `context` is threaded through all the calls.

This value is passed as the argument to every `:before` and `:after` function and it is returned by each function, possibly modified.

A `context` is a map with this structure:

```
{:coeffects {:event [:some-id :some-param]
             :db    <original contents of app-db>}

 :effects   {:db    <new value for app-db>
             :dispatch  [:an-event-id :param1]}

 :queue     <a collection of further interceptors>
 :stack     <a collection of interceptors already walked>}
```

`context` has `:coeffects` and `:effects` which, if this was a web server, would be somewhat analogous to `request` and `response` respectively.

`:coeffects` will contain the inputs required by the event handler (sitting presumably on the end of the chain). So that's data like the `:event` being processed, and the initial state of `db`.

The handler-returned side effects are put into `:effects` including, but not limited to, new values for `db`.

The first few interceptors in a chain (inserted by `reg-event-db`) have `:before` functions which **prime** the `:coeffects` by adding in `:event`, and `:db`. Of course, other interceptors can add further to `:coeffects`. Perhaps the event handler needs data from localstore, or a random number, or a DataScript connection. Interceptors can build up `:coeffects`, via their `:before`.

Equally, some interceptors in the chain will have `:after` functions which process the side effects accumulated into `:effects` including, but not limited to, updates to app-db.

Self Modifying

Through both stages (before and after), `context` contains a `:queue` of interceptors yet to be processed, and a `:stack` of interceptors already done.

In advanced cases, these values can be modified by the Interceptor functions through which the `context` is threaded.

What I'm saying is that interceptors can be dynamically added and removed from the `:queue` by existing Interceptors.

Credit

> All truths are easy to understand once they are discovered
> -- Galileo Galilei
>
> Things always become obvious after the fact
> -- Nassim Nicholas Taleb

This elegant and flexible arrangement was originally designed by the talented Pedestal Team. Thanks!

Write An Interceptor

Dunno about you, but I'm easily offended by underscores.

If we had a view which did this:

```
(dispatch [:delete-item 42])
```

We'd have to write this handler:

```
(reg-event-db
  :delete-item
  (fn
     [db [_ key-to-delete]]     ;;  <---- Arrgggghhh underscore
     (dissoc db key-to-delete)))
```

Do you see it there? In the event destructuring!!! Almost mocking us with that passive aggressive, understated thing it has going on!! Co-workers have said I'm "being overly sensitive", perhaps even pixel-ist, but you can see it, right? Of course you can.

What a relief it would be to not have it there, but how? We'll write an interceptor: `trim-event`

Once we have written `trim-event`, our registration will change to look like this:

```
(reg-event-db
  :delete-item
  [trim-event]                ;;  <--- interceptor added
  (fn
     [db [key-to-delete]]     ;;  <---yaaah!  no leading underscore
     (dissoc db key-to-delete)))
```

`trim-event` will need to change the `:coeffects` map (within `context`). Specifically, it will be changing the `:event` value within the `:coeffects`.

`:event` will start off as `[:delete-item 42]`, but will end up `[42]`. `trim-event` will remove that leading `:delete-item` because, by the time the event is being processed, we already know what id it has.

And, here it is:

```
(def trim-event
  (re-frame.core/->interceptor
    :id      :trim-event
    :before  (fn [context]
               (let [trim-fn (fn [event] (-> event rest vec))]
                 (update-in context [:coeffects :event] trim-fn)))))
```

As you read this, look back to what a `context` looks like.

Notes:

1. We use `->interceptor` to create an interceptor (which is just a map)
2. Our interceptor only has a `:before` function
3. Our `:before` is given `context` . It modifies it and returns it.
4. There is no `:after` for this Interceptor. It has nothing to do with the backwards processing flow of `:effects` . It is concerned only with `:coeffects` in the forward flow.

Wrapping Handlers

We're going well. Let's do an advanced wrapping.

Earlier, in the "Handlers Are Interceptors Too" section, I explained that `event handlers` are wrapped in an Interceptor and placed on the end of an Interceptor chain. Remember the whole `[std1 std2 in1 in2 h]` thing?

We'll now look at the `h` bit. How does an event handler get wrapped to be an Interceptor?

Reminder - there's two kinds of handler:

- the `-db` variety registered by `reg-event-db`
- the `-fx` variety registered by `reg-event-fx`

I'll now show how to wrap the `-db` variety.

Reminder: here's what a `-db` handler looks like:

```
(fn [db event]               ;; takes two params
  (assoc db :flag true))     ;; returns a new db
```

So, we'll be writing a function which takes a `-db` handler and returns an Interceptor which wraps that handler:

```
(defn db-handler->interceptor
  [db-handler-fn]
  (re-frame.core/->interceptor     ;; a utility function supplied by re-frame
    :id     :db-handler            ;; ids are decorative only
    :before (fn [context]
              (let [{:keys [db event]} (:coeffects context)    ;; extract db and event from coeffects
                    new-db (db-handler-fn db event)]           ;; call the handler
                 (assoc-in context [:effects :db] new-db)))))) ;; put db back into :effects
```

Notes:

1. Notice how this wrapper extracts data from the `context's` `:coeffects` and then calls the handler with that data (a handler must be called with `db` and `event`)
2. Equally notice how this wrapping takes the return value from the `-db` handler and puts it into `context's` `:effects`
3. The modified `context` (it has a new `:effects`) is returned
4. This is all done in `:before` . There is no `:after` (it is a noop). But this could have been reversed with the work happening in `:after` and `:before` a noop. Shrug. Remember that this Interceptor will be on the end of a chain.

Feeling confident? Try writing the wrapper for `-fx` handlers - it is just a small variation.

Summary

In this tutorial, we've learned:

1. When you register an event handler, you can supply a collection of interceptors:

```
 (reg-event-db
    :some-id
    [in1 in2]       ;; <--- a chain of 2 interceptors
    (fn [db v]      ;; <-- real handler here
       ....)))
```

2. When you are registering an event handler, you are associating an event id with a chain of interceptors including:

- the ones you supply (optional)
- an extra one on the end, which wraps the handler itself
- a couple at the beginning of the chain, put there by the `reg-event-db` or `reg-event-fx` .

3. An Interceptor Chain is executed in two stages. First a forwards sweep in which all `:before` functions are called, and then second, a backwards sweep in which the `:after` functions are called. A `context` will be threaded through all these calls.

4. Interceptors do interesting things:

- add to coeffects (data inputs to the handler)
- process side effects (returned by a handler)
- produce logs
- further process

In the next Tutorial, we'll look at (side) Effects in more depth. Later again, we'll look at Coeffects.

Appendix

The Built-in Interceptors

re-frame comes with some built-in Interceptors:

- **debug**: log each event as it is processed. Shows incremental `clojure.data/diff` reports.
- **trim-v**: a convenience. More readable handlers.

And some Interceptor factories (functions that return Interceptors):

- **enrich**: perform additional computations (validations?), after the handler has run. More derived data flowing.
- **after**: perform side effects, after a handler has run. Eg: use it to report if the data in `app-db` matches a schema.
- **path**: a convenience. Simplifies our handlers. Acts almost like `update-in` .

In addition, a Library like re-frame-undo provides an Interceptor factory called `undoable` which checkpoints app state.

To use them, first require them:

```
(ns my.core
  (:require
    [re-frame.core :refer [debug path]])
```

Previous: Effectful Handlers   Up: Index   Next: Effects

Effects

About 10% of the time, event handlers need to cause side effects.

This tutorial explains how side effects are actioned, how you can create your own side effects, and how you can make side effects a noop in event replays.

Table Of Contents

Where Effects Come From

When an event handler is registered via `reg-event-fx`, it must return effects.

Like this:

```
(reg-event-fx              ;; -fx registration, not -db registration
  :my-event
  (fn [cofx [_ a]]        ;; 1st argument is coeffects, instead of db
    {:db       (assoc (:db cofx) :flag  a)
     :dispatch [:do-something-else 3]}))   ;; return effects
```

`-fx` handlers return a description of the side-effects required, and that description is a map.

The Effects Map

An effects map contains instructions.

Each key/value pair in the map is one instruction - the `key` uniquely identifies the particular side effect required, and the `value` for that `key` provides further data. The structure of `value` is different for each side-effect.

Here's the two instructions from the example above:

```
{:db       (assoc db :flag  a)         ;; side effect on app-db
 :dispatch [:do-something-else 3]}     ;; dispatch this event
```

The `:db` `key` instructs that "app-db" should be `reset!` to the `value` supplied.

And the `:dispatch` `key` instructs that an event should be dispatched. The `value` is the vector to dispatch.

There's many other possible effects, like for example `:dispatch-later` or `:set-local-store`.

And so on. And so on. Which brings us to a problem.

Infinite Effects

While re-frame supplies a number of builtin effects, the set of possible effects is open ended.

What if you use PostgreSQL and want an effect which issues mutating queries? Or what if you want to send logs to Logentries or metrics to DataDog. Or write values to windows.location. And what happens if your database is X, Y or Z?

The list of effects is long and varied, with everyone needing to use a different combination.

So effect handling has to be extensible. You need a way to define your own side effects.

Extensible Side Effects

re-frame provides a function `reg-fx` through which you can register your own `Effect Handlers` .

Use it like this:

```
(reg-fx         ;; <-- registration function
   :butterfly   ;;  <1>
   (fn [value]  ;;  <2>
      ...
      ))
```

<1> the key for the effect. When later an effects map contains the key `:butterfly` , the function we are registering will be used to action it.

<2> the function which actions the side effect. Later, it will be called with one argument - the value in the effects map, for this key.

So, if an event handler returned these two effects:

```
{:dispatch   [:save-maiden 42]
 :butterfly  "Flapping"}         ;; butterfly effect, but no chaos !!
```

Then the function we registered for `:butterfly` would be called to handle that effect. And it would be called with the parameter "Flapping".

So, terminology:

- `:butterfly` is an "effect key"
- and the function registered is an "effect handler".

So re-frame has both `event` handlers and `effect` handlers and they are different, despite them both starting with `e` and ending in `t` !!

Writing An Effect Handler

A word of advice - make them as simple as possible, and then simplify them further. You don't want them containing any fancy logic.

Why? Well, because they are all side-effecty they will be a pain to test rigorously. And the combination of fancy logic and limited testing always ends in tears. If not now, later.

A second word of advice - when you create an effect handler, you also have to design (and document!) the structure of the `value` expected.

When you do, realise that you are designing a nano DSL for `value` and try to make that design simple too. If you resist being terse and smart, and instead, favor slightly verbose and obvious, your future self will thank you. Create as little cognitive overhead as possible for the eventual readers of your effectful code.

This advice coming from the guy who named effects `fx` ... Oh, the hypocrisy.

In my defence, here's the built-in effect handler for `:db` :

```
(reg-fx
  :db
  (fn [value]
    (reset! re-frame.db/app-db value)))
```

So, yeah, simple ... and, because of it, I can almost guarantee there's no bug in ... bang, crash, smoke, flames.

> Note: the return value of an effect handler is ignored.

:db Not Always Needed

An effects map does not need to include the `effect key` `:db`.

It is perfectly valid for an event handler to not change `app-db`.

In fact, it is perfectly valid for an event handler to return an effects map of `{}`. Slightly puzzling, but not a problem.

What Makes This Work?

A silently inserted interceptor.

Whenever you register an event handler via **either** `reg-event-db` or `reg-event-fx`, an interceptor, cunningly named `do-fx`, is inserted at the beginning of the chain.

Example: if your event handler registration looked like this:

```
(reg-event-fx
  :some-id
  [debug (path :right)]     ;; <-- two interceptors, apparently
  (fn [cofx _]
     {})                    ;; <-- imagine returned effects here
```

While it might look like you have registered with 2 interceptors, `reg-event-fx` will make it 3:

```
[do-fx debug (path :right)]
```

It silently inserts `do-fx` at the front, and this is a good thing.

The placement of `do-fx` at the beginning of the interceptor chain means its `:after` function would be the final act when the chain is executed (forwards and then backwards, as described in the Interceptor Tutorial).

In this final act, the `:after` function extracts `:effects` from `context` and simply iterates across the key/value pairs it contains, calling the registered "effect handlers" for each.

> For the record, the FISA Court requires that we deny all claims that `do-fx` is secretly injected NSA surveillance-ware.
> We also note that you've been particularly sloppy with your personal grooming today, including that you forgot to clean your teeth. Again.

If ever you want to take control of the way effect handling is done, create your own alternative to `reg-event-fx` and, in it, inject your own version of the `do-fx` interceptor at the front of the interceptor chain. It is only a few lines of code.

Order Of Effects?

There isn't one.

`do-fx` does not currently provide you with control over the order in which side effects occur. The `:db` side effect might happen before `:dispatch`, or not. You can't rely on it.

Note: if you feel you need ordering, then please open an issue and explain the usecase. The current absence of good usecases is the reason ordering isn't implemented. So give us a usercase and we'll revisit, maybe.

Further Note: if later ordering was needed, it might be handled via metadata on `:effects`. Also, perhaps by allowing `reg-fx` to optionally take two functions:

- an effects pre-process fn <-- new. Takes `:effects` returns `:effects`
- the effects handler (as already described above).

Anyway, these are all just possibilities. But not needed or implemented yet.

Effects With No Data

Some effects have no associated data:

```
(reg-event-fx
  :some-id
  (fn [coeffect _]
     {:exit-fullscreen nil}))    ;;   <--- no data, use a nil
```

In these cases, although it looks odd, just supply `nil` as the value for this key.

The associated effect handler would look like:

```
(reg-fx
  :exit-fullscreen
  (fn [_]             ;; we don't bother with that nil value
     (.exitFullscreen js/document)))
```

Testing And Noops

When you are running tests or replaying events, it is sometimes useful to stub out effects.

This is easily done - you simply register a noop effect handler.

Want to stub out the `:dispatch` effect? Do this:

```
(reg-fx
  :dispatch
  (fn [_] ))    ;; a noop
```

If your test does alter registered effect handlers, and you are using `cljs.test`, then you can use a `fixture` to restore all effect handlers at the end of your test:

```
(defn fixture-re-frame
  []
  (let [restore-re-frame (atom nil)]
    {:before #(reset! restore-re-frame (re-frame.core/make-restore-fn))
     :after  #(@restore-re-frame)}))

(use-fixtures :each (fixture-re-frame))
```

`re-frame.core/make-restore-fn` creates a checkpoint for re-frame state (including registered handlers) to which you can return.

Existing Effect Handlers

Built-in effect handlers are detailed the API document.

And please review the External-Resources document for a list of 3rd party Effect Handlers.

Summary

The 4 Point Summary in note form:

1. Event handlers should only return effect declaratively
2. They return a map like `{:effect1 value1 :effect2 value2}`
3. Keys of this map can refer to builtin effects handlers (see below) or custom ones
4. We use `reg-fx` to register our own effects handlers, built-in ones are already registered

---

Previous: Interceptors Up: Index Next: Coeffects

Coeffects

This tutorial explains `coeffects`.

It explains what they are, how they can be "injected", and how to manage them in tests.

Table Of Contents

What Are They?

`coeffects` are the data resources that an event handler needs to perform its computation.

Because the majority of event handlers only need `db` and `event`, there's a specific registration function, called `reg-event-db`, which delivers ONLY these two coeffects as arguments to an event handler, making this common case easy to program.

But sometimes an event handler needs other data inputs to perform its computation. Things like a random number, or a GUID, or the current datetime. Perhaps it needs access to a DataScript connection.

An Example

This handler obtains data directly from LocalStore:

```
(reg-event-db
   :load-defaults
   (fn [db _]
     (let [val (js->clj (.getItem js/localStorage "defaults-key"))]  ;; <-- Problem
       (assoc db :defaults val))))
```

This works, but there's a cost.

Because it has directly accessed LocalStore, this event handler is not pure, and impure functions cause well-documented paper cuts.

How We Want It

Our goal in this tutorial will be to rewrite this event handler so that it **only** uses data from arguments. This will take a few steps.

The first is that we switch to using `reg-event-fx` (instead of `reg-event-db`).

Event handlers registered via `reg-event-fx` are slightly different to those registered via `reg-event-db`. `-fx` handlers get two arguments, but the first is not `db`. Instead it is an argument which we will call `cofx` (that's a nice distinct name which will aid communication).

Previous tutorials showed there's a `:db` key in `cofx`. We now want `cofx` to have other keys and values, like this:

```
(reg-event-fx                     ;; note: -fx
   :load-defaults
   (fn [cofx event]                 ;; cofx means coeffects
     (let [val (:local-store cofx)  ;; <-- get data from cofx
           db  (:db cofx)]          ;; <-- more data from cofx
       {:db (assoc db :defaults val))})) ;; returns an effect
```

Notice how `cofx` magically contains a `:local-store` key with the right value. Nice! But how do we make this magic happen?

Abracadabra

Each time an event handler is executed, a brand new `context` (map) is created, and within that `context` is a `:coeffects` key which is a further map (initially empty).

That pristine `context` value (containing a pristine `:coeffects` map) is threaded through a chain of Interceptors before it finally reaches our event handler, which sits on the end of the chain, itself wrapped up in an interceptor. We know this story well from a previous tutorial.

So, all members of the Interceptor chain have the opportunity to `assoc` into `:coeffects` within their `:before` function, cumulatively building up what it holds. Later, our event handler, which sits on the end of the chain, magically finds just the right data (like a value for the key `:local-store`) in its first `cofx` argument. So, it is the event handler's Interceptors which put it there.

Which Interceptors?

If Interceptors put data in `:coeffects`, then we'll need to add the right ones when we register our event handler.

Something like this (this handler is the same as before, except for one detail):

```
(reg-event-fx
   :load-defaults
   [ (inject-cofx :local-store "defaults-key") ]     ;; <-- this is new
   (fn [cofx event]
     (let [val (:local-store cofx)
           db  (:db cofx)]
       {:db (assoc db :defaults val))}))
```

Look at that - my event handler has a new Interceptor! It is injecting the right key/value pair (`:local-store`) into `context's` `:coeffects`, which itself then goes on to be the first argument to our event handler (`cofx`).

`inject-cofx`

`inject-cofx` is part of the re-frame API.

It is a function which returns an Interceptor whose `:before` function loads a key/value pair into a `context's` `:coeffects` map.

`inject-cofx` takes either one or two arguments. The first is always the `id` of the coeffect required (called a `cofx-id`). The 2nd is an optional addition value.

So, in the case above, the `cofx-id` was `:local-store` and the additional value was "defaults-key" which was presumably the LocalStore key.

More `inject-cofx`

Here's some other usage examples:

- `(inject-cofx :random-int 10)`
- `(inject-cofx :guid)`
- `(inject-cofx :now)`

I could create an event handler which has access to 3 coeffects:

```
(reg-event-fx
    :some-id
    [(inject-cofx :random-int 10) (inject-cofx :now)  (inject-cofx :local-store "blah")]  ;; 3
    (fn [cofx _]
       ... in here I can access cofx's keys :now :local-store and :random-int))
```

But that's probably just greedy, and not very useful.

And so, to the final piece in the puzzle: how does `inject-cofx` know what to do when it is given `:now` or `:local-store` ? Each `cofx-id` requires a different action.

Meet `reg-cofx`

This function is also part of the re-frame API.

It allows you to associate a `cofx-id` (like `:now` or `:local-store`) with a handler function that injects the right key/value pair.

The function you register will be passed two arguments:

- a `:coeffects` map (to which it should add a key/value pair), and
- optionally, the additional value supplied to `inject-cofx`

and it is expected to return a modified `:coeffects` map.

Example Of `reg-cofx`

Above, we wrote an event handler that wanted `:now` data to be available. Here is how a handler could be registered for `:now` :

```
(reg-cofx               ;; registration function
   :now                 ;; what cofx-id are we registering
   (fn [coeffects _]    ;; second parameter not used in this case
      (assoc coeffects :now (js.Date.))))   ;; add :now key, with value
```

The outcome is:

1. because that cofx handler above is now registered for `:now` , I can
2. add an Interceptor to an event handler which
3. looks like `(inject-cofx :now)`
4. which means within that event handler I can access a `:now` value from `cofx`

As a result, my event handler is pure.

Another Example Of `reg-cofx`

This:

```
(reg-cofx               ;; new registration function
   :local-store
   (fn [coeffects local-store-key]
      (assoc coeffects
             :local-store
             (js->clj (.getItem js/localStorage local-store-key)))))
```

With these two registrations in place, I could now use both `(inject-cofx :now)` and `(inject-cofx :local-store "blah")` in an event handler's interceptor chain.

To put this another way: I can't use `(inject-cofx :blah)` UNLESS I have previously used `reg-cofx` to register a handler for `:blah` . Otherwise `inject-cofx` doesn't know how to inject a `:blah` .

Secret Interceptors

In a previous tutorial we learned that `reg-events-db` and `reg-events-fx` add Interceptors to the front of any chain during registration. We found they inserted an Interceptor called `do-fx` .

I can now reveal that they also add `(inject-cofx :db)` at the front of each chain.

Guess what that injects into the `:coeffects` of every event handler? This is how `:db` is always available to event handlers.

Okay, so that was the last surprise. Now you know everything.

If ever you wanted to use DataScript, instead of an atom-containing-a-map like `app-db` , you'd replace `reg-event-db` and `reg-event-fx` with your own registration functions and have them auto insert the DataScript connection.

Testing

During testing, you may want to stub out certain coeffects.

You may, for example, want to test that an event handler works using a specific `now` .

In your test, you'd mock out the cofx handler:

```
(reg-cofx
   :now
   (fn [coeffects _]
      (assoc coeffects :now (js/Date. 2016 1 1)))   ;; now was then
```

If your test does alter registered coeffect handlers, and you are using `cljs.test` , then you can use a `fixture` to restore all coeffects at the end of your test:

```
(defn fixture-re-frame
  []
  (let [restore-re-frame (atom nil)]
    {:before #(reset! restore-re-frame (re-frame.core/make-restore-fn))
     :after  #(@restore-re-frame)}))

(use-fixtures :each (fixture-re-frame))
```

`re-frame.core/make-restore-fn` creates a checkpoint for re-frame state (including registered handlers) to which you can return.

The 5 Point Summary

In note form:

1. Event handlers should only source data from their arguments
2. We want to "inject" required data into the first, cofx argument
3. We use the `(inject-cofx :key)` interceptor in registration of the event handler
4. It will look up the registered cofx handler for that `:key` to do the injection
5. We must have previously registered a cofx handler via `reg-cofx`

Previous: Effects   Up: Index   Next: Infographic

Subscription Infographic

There's two things to do here.

First, please read through the annotated subscription code in the todomvc example.

Then, look at this Infographic:

Previous: Coeffects Up: Index Next: Correcting a wrong

Subscriptions Cleanup

There's a problem and we need to fix it.

The Problem

The simple example, used in the earlier code walk through, is not idomatic re-frame. It has a flaw.

It does not obey the re-frame rule: **keep views as simple as possible**.

A view shouldn't do any computation on input data. Its job is just to compute hiccup. The subscriptions it uses should deliver the data already in the right structure, ready for use in hiccup generation.

Just Look

Here be the horrors:

```
(defn clock
  []
  [:div.example-clock
   {:style {:color @(rf/subscribe [:time-color])}}
   (-> @(rf/subscribe [:time])
       .toTimeString
       (clojure.string/split " ")
       first)])
```

That view obtains data from a `[:time]` subscription and then it massages that data into the form it needs for use in the hiccup. We don't like that.

The Solution

Instead, we want to use a new `[:time-str]` subscription which will deliver the data all ready to go, so the view is 100% concerned with hiccup generation only. Like this:

```
(defn clock
  []
  [:div.example-clock
   {:style {:color @(rf/subscribe [:time-color])}}
   @(rf/subscribe [:time-str])])
```

Which, in turn, means we must write this `time-str` subscription handler:

```
(reg-sub
  :time-str
  (fn [_ _]
    (subscribe [:time]))
  (fn [t _]
    (-> t
       .toTimeString
       (clojure.string/split " ")
       first)))
```

Much better.

You'll notice this new subscription handler belongs to the "Level 3" layer of the reactive flow. See the Infographic.

Another Technique

Above, I suggested this:

```
(defn clock
  []
  [:div.example-clock
   {:style {:color @(rf/subscribe [:time-color])}}
   @(rf/subscribe [:time-str])])
```

But that may offend your aesthetics. Too much noise with those two `@` ?

To clean this up, we can define a new `listen` function:

```
(defn listen
  [query-v]
  @(rf/subscribe query-v))
```

And then rewrite:

```
(defn clock
  []
  [:div.example-clock
   {:style {:color (listen [:time-color])}}
   (listen [:time-str])])
```

So, at the cost of writing your own function, `listen` , the code is now less noisy AND there's less chance of us forgetting an `@` (which can lead to odd problems).

LambdaIsland Naming (LIN)

I've ended up quite liking the alternative names suggested by Lambda Island Videos:

```
(def <sub (comp deref re-frame.core/subscribe))   ;; aka listen (above)
(def >evt re-frame.core/dispatch)
```

Say It Again

So, if, in code review, you saw this view function:

```
(defn show-items
  []
  (let [sorted-items (sort @(subscribe [:items]))]
    (into [:div] (for [i sorted-items] [item-view i]))))
```

What would you (supportively) object to?

That `sort` , right? Computation in the view. Instead, we want exactly the right data delivered to the view - no further computation required - the view's job is to simply make `hiccup` .

The solution is to create a subscription that delivers items already sorted.

```
(reg-sub
   :sorted-items
   (fn [_ _]  (subscribe [:items]))
   (fn [items _]
      (sort items))
```

Now, in this case the computation is a bit trivial, but the moment it is a little tricky, you'll want to test it. So separating it out from the view will make that easier.

To make it testable, you may structure like this:

```
(defn item-sorter
  [items _]
  (sort items))

(reg-sub
   :sorted-items
   (fn [_ _]  (subscribe [:items]))
   item-sorter)
```

Now it is easy to test `item-sorter` independently.

And There's Another Benefit

re-frame de-duplicates signal graph nodes.

If, for example, two views wanted to `(subscribe [:sorted-items])` only the one node (in the signal graph) would be created. Only one node would be doing that potentially expensive sorting operation (when items changed) and values from it would be flowing through to both views.

That sort of efficiency can't happen if this views themselves are doing the `sort`.

de-duplication

As I described above, two, or more, concurrent subscriptions for the same query will source reactive updates from the one executing handler - from the one node in the signal graph.

How do we know if two subscriptions are "the same"? Answer: two subscriptions are the same if their query vectors test `=` to each other.

So, these two subscriptions are *not* "the same": `[:some-event 42]` `[:some-event "blah"]`. Even though they involve the same event id, `:some-event`, the query vectors do not test `=`.

This feature shakes out nicely because re-frame has a data oriented design.

A Final FAQ

The following issue comes up a bit.

You will end up with a bunch of level 1 `reg-sub` which look the same (they directly extract a path within `app-db`):

```
(reg-sub
   :a
   (fn [db _]
     (:a db)))
```

```
(reg-sub
   :b
   (fn [db _]
     (-> db :top :b)))
```

Now, you think and design abstractly for a living, and that repetition will feel uncomfortable. It will call to you like a Siren: "refaaaaactoooor meeeee". "Maaaake it DRYYYY". So here's my tip: tie yourself to the mast and sail on. That repetition is good. It is serving a purpose. Just sail on.

The WORST thing you can do is to flex your magnificent abstraction muscles and create something like this:

```
(reg-sub
   :extract-any-path
```

```
    (fn [db path]
      (get-in db path))
```

"Genius!", you think to yourself. "Now I only need one direct `reg-sub` and I supply a path to it. A read-only cursor of sorts. Look at the code I can delete."

Neat and minimal it most certainly is, yes, but genius it isn't. You are now asking the code USING the subscription to provide the path. You have traded some innocuous repetition for longer term fragility, and that's not a good trade.

What fragility? Well, the view which subscribes using, say, `(subscribe [:extract-any-path [:a]])` now "knows" about (depends on) the structure within `app-db`.

What happens when you inevitably restructure `app-db` and put that `:a` path under another high level branch of `app-db`? You will have to run around all the views, looking for the paths supplied, knowing which to alter and which to leave alone. Fragile.

We want our views to declaratively ask for data, but they should have no idea where it comes from. It is the job of a subscription to know where data comes from.

Remember our rule at the top: **keep views as simple as possible**. Don't give them knowledge or tasks outside their remit.

Previous: Infographic   Up: Index   Next: Flow Mechanics

> In a rush? You can get away with skipping this page on the first pass.
> Next page: Basic App Structure

Flow Mechanics

This tutorial explains the underlying reactive mechanism used in dominoes 4-5-6. It goes on to introduce `re-frame.core/reg-sub-raw` .

On Flow

Arguments from authority ...

> Everything flows, nothing stands still. (Panta rhei)
>
> No man ever steps in the same river twice for it's not the same river and he's not the same man.

Heraclitus 500 BC. Who, being Greek, had never seen a frozen river. alt version.

> Think of an experience from your childhood. Something you remember clearly, something you can see, feel, maybe even smell, as if you were really there. After all you really were there at the time, weren't you? How else could you remember it? But here is the bombshell: you weren't there. Not a single atom that is in your body today was there when that event took place .... Matter flows from place to place and momentarily comes together to be you. Whatever you are, therefore, you are not the stuff of which you are made. If that does not make the hair stand up on the back of your neck, read it again until it does, because it is important.

Steve Grand

How Flow Happens In Reagent

To implement a reactive flow, Reagent provides a `ratom` and a `reaction` . re-frame uses both of these building blocks, so let's now make sure we understand them.

`ratoms` behave just like normal ClojureScript atoms. You can `swap!` and `reset!` them, `watch` them, etc.

From a ClojureScript perspective, the purpose of an atom is to hold mutable data. From a re-frame perspective, we'll tweak that paradigm slightly and **view a `ratom` as having a value that changes over time.** Seems like a subtle distinction, I know, but because of it, re-frame sees a `ratom` as a Signal.

A Signal is a value that changes over time. So it is a stream of values. Each time a ratom gets `reset!` that's a new value in the stream.

The 2nd building block, `reaction` , acts a bit like a function. It's a macro which wraps some `computation` (a block of code) and returns a `ratom` holding the result of that `computation` .

The magic thing about a `reaction` is that the `computation` it wraps will be automatically re-run whenever 'its inputs' change, producing a new output (return) value.

Eh, how?

Well, the `computation` is just a block of code, and if that code dereferences one or more `ratoms` , it will be automatically re-run (recomputing a new return value) whenever any of these dereferenced `ratoms` change.

To put that yet another way, a `reaction` detects a `computation's` input Signals (aka input `ratoms`) and it will `watch` them, and when, later, it detects a change in one of them, it will re-run that computation, and it will `reset!` the new result of that computation into the `ratom` originally returned.

So, the `ratom` returned by a `reaction` is itself a Signal. Its value will change over time when the `computation` is re-run.

So, via the interplay between `ratoms` and `reactions` , values 'flow' into computations and out again, and then into further computations, etc. "Values" flow (propagate) through the Signal graph.

But this Signal graph must be without cycles, because cycles cause mayhem! re-frame achieves a unidirectional flow.

Right, so that was a lot of words. Some code to clarify:

```
(ns example1
 (:require-macros [reagent.ratom :refer [reaction]])  ;; reaction is a macro
 (:require        [reagent.core  :as    reagent]))

(def app-db  (reagent/atom {:a 1}))           ;; our root ratom  (signal)

(def ratom2  (reaction {:b (:a @app-db)}))    ;; reaction wraps a computation, returns a signal
(def ratom3  (reaction (condp = (:b @ratom2)  ;; reaction wraps another computation
                             0 "World"
                             1 "Hello")))

;; Notice that both computations above involve de-referencing a ratom:
;;   - app-db in one case
;;   - ratom2 in the other
;; Notice that both reactions above return a ratom.
;; Those returned ratoms hold the (time varying) value of the computations.

(println @ratom2)    ;; ==>  {:b 1}       ;; a computed result, involving @app-db
(println @ratom3)    ;; ==> "Hello"       ;; a computed result, involving @ratom2

(reset!  app-db  {:a 0})       ;; this change to app-db, triggers re-computation
                               ;; of ratom2
                               ;; which, in turn, causes a re-computation of ratom3

(println @ratom2)    ;; ==>  {:b 0}    ;; ratom2 is result of {:b (:a @app-db)}
(println @ratom3)    ;; ==> "World"    ;; ratom3 is automatically updated too.
```

So, in FRP-ish terms, a `reaction` will produce a "stream" of values over time (it is a Signal), accessible via the `ratom` it returns.

Components (view functions)

When using Reagent, your primary job is to write one or more `components`. This is the view layer.

Think about `components` as `pure functions` - data in, Hiccup out. `Hiccup` is ClojureScript data structures which represent DOM. Here's a trivial component:

```
(defn greet
  []
  [:div "Hello ratoms and reactions"])
```

And if we call it:

```
(greet)
;; ==>  [:div "Hello ratoms and reactions"]
```

You'll notice that our component is a regular Clojure function, nothing special. In this case, it takes no parameters and it returns a ClojureScript vector (formatted as Hiccup).

Here is a slightly more interesting (parameterised) component (function):

```
(defn greet                    ;; greet has a parameter now
  [name]                       ;; 'name' is a ratom  holding a string
  [:div "Hello "  @name])      ;; dereference 'name' to extract the contained value

;; create a ratom, containing a string
(def n (reagent/atom "re-frame"))

;; call our `component` function, passing in a ratom
(greet n)
;; ==>  [:div "Hello " "re-frame"]    returns a vector
```

So components are easy - at core they are a render function which turns data into Hiccup (which will later become DOM).

Now, let's introduce `reaction` into this mix. On the one hand, I'm complicating things by doing this, because Reagent allows you to be ignorant of the mechanics I'm about to show you. (It invisibly wraps your components in a `reaction` allowing you to be blissfully ignorant of how the magic happens.)

On the other hand, it is useful to understand exactly how the Reagent Signal graph is wired.

```
(defn greet                ;; a component - data in, Hiccup out.
  [name]                   ;; name is a ratom
  [:div "Hello "  @name])  ;; dereference name here, to extract the value within

(def n (reagent/atom "re-frame"))

;; The computation '(greet n)' returns Hiccup which is stored into 'hiccup-ratom'
(def hiccup-ratom  (reaction (greet n)))    ;; <-- use of reaction !!!

;; what is the result of the initial computation ?
(println @hiccup-ratom)
;; ==>  [:div "Hello " "re-frame"]    ;; returns hiccup  (a vector of stuff)

;; now change 'n'
;; 'n' is an input Signal for the reaction above.
;; Warning: 'n' is not an input signal because it is a parameter. Rather, it is
;; because 'n' is dereferenced within the execution of the reaction's computation.
;; reaction notices what ratoms are dereferenced in its computation, and watches
;; them for changes.
(reset! n "blah")            ;;    n changes

;; The reaction above will notice the change to 'n' ...
;; ... and will re-run its computation ...
;; ... which will have a new "return value"...
;; ... which will be "reset!" into "hiccup-ratom"
(println @hiccup-ratom)
;; ==>   [:div "Hello " "blah"]    ;; yep, there's the new value
```

So, as `n` changes value over time (via a `reset!`), the output of the computation `(greet n)` changes, which in turn means that the value in `hiccup-ratom` changes. Both `n` and `hiccup-ratom` are FRP Signals. The Signal graph we created causes data to flow from `n` into `hiccup-ratom`.

Derived Data, flowing.

Truth Interlude

I haven't been entirely straight with you:

1. Reagent re-runs `reactions` (re-computations) via requestAnimationFrame. So a re-computation happens about 16ms after an input Signals change is detected, or after the current thread of processing finishes, whichever is the greater. So if you are in a bREPL and you run the lines of code above one after the other too quickly, you might not see the re-computation done immediately after `n` gets reset!, because the next animationFrame hasn't run (yet). But you could add a `(reagent.core/flush)` after the reset! to force re-computation to happen straight away.

2. `reaction` doesn't actually return a `ratom`. But it returns something that has ratom-nature, so we'll happily continue believing it is a `ratom` and no harm will come to us.

On with the rest of my lies and distortions...

reg-sub-raw

This low level part of the API provides a way to register a subscription handler - so the intent is similar to `reg-sub`.

You use it like other registration functions:

```
(re-frame.core/reg-sub-raw   ;; it is part of the API
  :query-id     ;; later use (subscribe [:query-id])
  some-fn)      ;; this function provides the reactive stream
```

The interesting bit is how `some-fn` is written. Here's an example:

```
(defn some-fn
  [app-db event]    ;; app-db is not a value, it is a reagent/atom
  (reaction (get-in @app-db [:some :path])))  ;; returns a reaction
```

Notice:

1. `app-db` is a reagent/atom. It is not a value like `reg-sub` gets.
2. it returns a `reaction` which does a computation. It does not return a value like `reg-sub` does.
3. Within that `reaction` `app-db` is deref-ed (see use of `@`)

As a result of point 3, each time `app-db` changes, the wrapped `reaction` will rerun. `app-db` is an input signal to that `reaction`.

Unlike `reg-sub`, there is no 3-arity version of `reg-sub-raw`, so there's no way for you to provide an input signals function. Instead, even simpler, you can just use `subscribe` within the `reaction` itself. For example:

```
(defn some-fn
   [app-db event]
   (reaction
     (let [a-path-element @(subscribe [:get-path-part])]   ;; <-- subscribe used here
       (get-in @app-db [:some a-path-element]))))
```

As you can see, this `reaction` has two input signals: `app-db` and `(subscribe [:get-path-part])`. If either changes, the `reaction` will rerun.

In some cases, the returned `reaction` might not even use `app-db` and, instead, it might only use `subscribe` to provide input signals. In that case, the registered subscription would belong to "Level 3" of the signal graph (discussed in earlier tutorials).

Remember to deref any use of `app-db` and `subscribe`. It is a rookie mistake to forget. I do it regularly.

Instead of using `reaction` (a macro), you can use `reagent/make-reaction` (a utility function) which gives you the additional ability to attach an `:on-dispose` handler to the returned reaction, allowing you to do cleanup work when the subscription is no longer needed. See an example of using `:on-dispose` here

Example reg-sub-raw

The following use of `reg-sub` can be found in the todomvc example:

```
(reg-sub
  :visible-todos

  ;; signal function - returns a vector of two input signals
  (fn [query-v _]
    [(subscribe [:todos])
     (subscribe [:showing])])

  ;; the computation function - 1st arg is a 2-vector of values
  (fn [[todos showing] _]
    (let [filter-fn (case showing
                      :active (complement :done)
                      :done   :done
                      :all    identity)]
      (filter filter-fn todos))))
```

we could rewrite this use of `reg-sub` using `reg-sub-raw` like this:

```
(reg-sub-raw
  :visible-todos
```

```
    (fn [app-db event]  ;; app-db not used, name shown for clarity
      (reaction         ;; wrap the computation in a reaction
        (let [todos   @(subscribe [:todos])   ;; input signal #1
              showing @(subscribe [:showing]) ;; input signal #2
              filter-fn (case showing
                          :active (complement :done)
                          :done   :done
                          :all    identity)]
          (filter filter-fn todos))))
```

A view could do `(subscribe [:visible-todos])` and never know which of the two variations above was used. Same result delivered.

Previous: Correcting a wrong   Up: Index   Next: Basic App Structure

Simpler Apps

To build a re-frame app, you:

- design your app's data structures (data layer)
- write Reagent view functions (domino 5)
- write event handler functions (control layer and/or state transition layer, domino 2)
- write subscription functions (query layer, domino 4)

For simpler apps, you should put code for each layer into separate files:

```
src
├── core.cljs         <--- entry point, plus history, routing, etc
├── db.cljs           <--- schema, validation, etc  (data layer)
├── views.cljs        <--- reagent views (view layer)
├── events.cljs       <--- event handlers (control/update layer)
└── subs.cljs         <--- subscription handlers  (query layer)
```

For a living example of this approach, look at the todomvc example.

No really, you should absolutely look at the todomvc example example, as soon as possible. It contains all sorts of tips.

There's A Small Gotcha

If you adopt this structure, there's a gotcha.

`events.cljs` and `subs.cljs` will never be `required` by any other namespaces. To the Google Closure dependency mechanism it appears as if these two namespaces are not needed and it doesn't load them.

And, if the code does not get loaded, the registrations in these namespaces never happen. You'll then be puzzled as to why none of your events handlers are registered.

Once you twig to what's going on, the solution is easy. You must explicitly `require` both namespaces, `events` and `subs`, in your `core` namespace. Then they'll be loaded and the registrations will occur as that loading happens.

Larger Apps

Assuming your larger apps have multiple "panels" (or "views") which are relatively independent, you might use this structure:

```
src
├── core.cljs             <--- entry point, plus history, routing, etc
├── panel-1
│   ├── db.cljs           <--- schema, validation, etc  (data layer)
│   ├── subs.cljs         <--- subscription handlers  (query layer)
│   ├── views.cljs        <--- reagent components (view layer)
│   └── events.cljs       <--- event handlers (control/update layer)
├── panel-2
│   ├── db.cljs           <--- schema, validation. etc  (data layer)
│   ├── subs.cljs         <--- subscription handlers  (query layer)
│   ├── views.cljs        <--- reagent components (view layer)
│   └── events.cljs       <--- event handlers (control/update layer)
.
.
└── panel-n
```

Previous: Correcting a wrong   Up: Index   Next: Navigation

Table Of Contents

What About Navigation?

How do I switch between different panels of a larger app?

Your `app-db` could have an `:active-panel` key containing an id for the panel being displayed.

When the user does something navigation-ish (selects a tab, a dropdown or something which changes the active panel), then the associated event and dispatch look like this:

```
(re-frame/reg-event-db
  :set-active-panel
  (fn [db [_ value]]
    (assoc db :active-panel value)))

(re-frame/dispatch
  [:set-active-panel :panel1])
```

A high level reagent view has a subscription to :active-panel and will switch to the associated panel.

```
(re-frame/reg-sub
  :active-panel
  (fn [db _]
    (:active-panel db)))

(defn panel1
 []
 [:div  {:on-click #(re-frame/dispatch [:set-active-panel :panel2])}
        "Here" ])

(defn panel2
 []
 [:div "There"])

(defn high-level-view
  []
  (let [active  (re-frame/subscribe [:active-panel])]
    (fn []
      [:div
       [:div.title   "Heading"]
       (condp = @active                ;; or you could look up in a map
         :panel1   [panel1]
         :panel2   [panel2])]))))
```

Continue to Namespaced Keywords to reduce clashes on ids.

Previous: Basic App Structure   Up: Index   Next: Namespaced Keywords

Table Of Contents

Namespaced Ids

As an app gets bigger, you'll tend to get clashes on ids - event-ids, or query-ids (subscriptions), etc.

One panel will need to `dispatch` an `:edit` event and so will another, but the two panels will have different handlers. So how then to not have a clash? How then to distinguish between one `:edit` event and another?

Your goal should be to use event-ids which encode both the event itself (`:edit` ?) and the context (`:panel1` or `:panel2` ?).

Luckily, ClojureScript provides a nice easy solution: use keywords with a **synthetic namespace**. Perhaps something like `:panel1/edit` and `:panel2/edit` .

You see, ClojureScript allows the namespace in a keyword to be a total fiction. I can have the keyword `:panel1/edit` even though `panel1.cljs` doesn't exist.

Naturally, you'll take advantage of this by using keyword namespaces which are both unique and descriptive.

Previous: Navigation Up: Index Next: Loading Initial Data

Bootstrapping Application State

To bootstrap a re-frame application, you need to:

1. register handlers:
   - subscription (via `reg-sub`)
   - events (via `reg-event-db` or `reg-event-fx`)
   - effects (via `reg-fx`)
   - coeffects (via `reg-cofx`)
2. kickstart reagent (views)
3. Load the right initial data into `app-db` which might, for example, be a `merge` of:
   - Some default values
   - Values stored in LocalStorage
   - Values obtained via service calls to server

Point 3 is the interesting bit and will be the main focus of this page, but let's work our way through them ...

1. Register Handlers

re-frame's various handlers all work in the same way. You declare and register your handlers in the one step, like this "event handler" example:

```
(re-frame/reg-event-db       ;; event handler will be registered automatically
  :some-id
  (fn [db [_ value]]
    ...  do some state change based on db and value ))
```

As a result, there's nothing further you need to do because handler registration happens as a direct result of loading the code (presumably via a `<script>` tag in your HTML file).

2. Kick Start Reagent

Create a function `main` which does a `reagent/render` of your root reagent component `main-panel` :

```
(defn main-panel       ;; my top level reagent component
  []
  [:div "Hello DDATWD"])

(defn ^:export main     ;; call this to bootstrap your app
  []
  (reagent/render [main-panel]
                  (js/document.getElementById "app")))
```

Mounting the top level component `main-panel` will trigger a cascade of child component creation. The full DOM tree will be rendered.

3. Loading Initial Data

Let's rewrite our `main-panel` component to use a subscription. In effect, we want it to source and render some data held in `app-db` .

First, we'll create the subscription handler:

```
(re-frame.core/reg-sub     ;; a new subscription handler
  :name               ;; usage (subscribe [:name])
  (fn [db _]
    (:display-name db)))  ;; extracts `:display-name` from app-db
```

And now we use that subscription:

```
(defn main-panel
  []
  (let [name  (re-frame.core/subscribe [:name])]  ;; <--- a subscription  <---
      [:div "Hello " @name])))   ;; <--- use the result of the subscription
```

The user of our app will see funny things if that `(subscribe [:name])` doesn't deliver good data. But how do we ensure "good data"?

That will require:

1. getting data into `app-db`; and
2. not get into trouble if that data isn't yet in `app-db`. For example, the data may have to come from a server and there's latency.

Note: `app-db` initially contains `{}`

Getting Data Into `app-db`

Only event handlers can change `app-db`. Those are the rules!! Indeed, even initial values must be put in `app-db` via an event handler.

Here's an event handler for that purpose:

```
(re-frame.core/reg-event-db
  :initialise-db                 ;; usage: (dispatch [:initialise-db])
  (fn [_ _]                         ;; Ignore both params (db and event)
     {:display-name "DDATWD"     ;; return a new value for app-db
      :items [1 2 3 4]}))
```

You'll notice that this handler does nothing other than to return a `map`. That map will become the new value within `app-db`.

We'll need to dispatch an `:initialise-db` event to get it to execute. `main` seems like the natural place:

```
(defn ^:export main
  []
  (re-frame.core/dispatch [:initialise-db])   ;;  <--- this is new
  (reagent/render [main-panel]
                  (js/document.getElementById "app")))
```

But remember, event handlers execute async. So although there's a `dispatch` within `main`, the event is simply queued, and the handler for `:initialise-db` will not be run until sometime after `main` has finished.

But how long after? And is there a race condition? The component `main-panel` (which assumes good data) might be rendered before the `:initialise-db` event handler has put good data into `app-db`.

We don't want any rendering (of `main-panel`) until after `app-db` has been correctly initialised.

Okay, so that's enough of teasing-out the issues. Let's see a quick sketch of the entire pattern. It is very straight-forward.

The Pattern

```
(re-frame.core/reg-sub   ;; supplied main-panel with data
  :name                  ;; usage (subscribe [:name])
  (fn  [db _]
    (:display-name db)))

(re-frame.core/reg-sub   ;; we can check if there is data
  :initialised?          ;; usage (subscribe [:initialised?])
  (fn  [db _]
    (not (empty? db))))  ;; do we have data

(re-frame.core/reg-event-db
   :initialise-db
   (fn [db _]
       (assoc db :display-name "Jane Doe")))

(defn main-panel    ;; the top level of our app
  []
  (let [name  (re-frame.core/subscribe [:name])]   ;; we need there to be good data
    [:div "Hello " @name])))

(defn top-panel    ;; this is new
  []
  (let [ready?  (re-frame.core/subscribe [:initialised?])]
    (if-not @ready?         ;; do we have good data?
      [:div "Initialising ..."]   ;; tell them we are working on it
      [main-panel])))      ;; all good, render this component

(defn ^:export main     ;; call this to bootstrap your app
  []
  (re-frame.core/dispatch [:initialise-db])
  (reagent/render [top-panel]
                  (js/document.getElementById "app")))
```

Scales Up

This pattern scales up easily.

For example, imagine a more complicated scenario in which your app is not fully initialised until 2 backend services supply data.

Your `main` might look like this:

```
(defn ^:export main     ;; call this to bootstrap your app
  []
  (re-frame.core/dispatch [:initialise-db])           ;; basics
  (re-frame.core/dispatch [:load-from-service-1])     ;; ask for data from service-1
  (re-frame.core/dispatch [:load-from-service-2])     ;; ask for data from service-2
  (reagent.core/render [top-panel]
                  (js/document.getElementById "app")))
```

Your `:initialised?` test then becomes more like this sketch:

```
(re-frame.core/reg-sub
  :initialised?          ;; usage (subscribe [:initialised?])
  (fn  [db _]
    (and  (not (empty? db))
          (:service1-answered? db)
          (:service2-answered? db)))))
```

This assumes boolean flags are set in `app-db` when data was loaded from these services.

Cheating - Synchronous Dispatch

In simple cases, you can simplify matters by using `dispatch-sync` (instead of `dispatch`) in the main function.

This technique can be seen in the TodoMVC Example.

`dispatch` queues an event for later processing, but `dispatch-sync` acts like a function call and handles an event immediately. That's useful for initial data load we are considering, particularly for simple apps. Using `dispatch-sync` guarantees that initial state will be in place before any views are mounted, so we know they'll subscribe to sensible values. We don't need a guard like `top-panel` (introduced above).

But don't get into the habit of using `dispatch-sync` everywhere. It is the right tool in this context and, sometimes, when writing tests, but `dispatch` is the staple you should use everywhere else.

Loading Initial Data From Services

Above, in our example `main`, we imagined using `(re-frame/dispatch [:load-from-service-1])` to request data from a backend services. How would we write the handler for this event?

The next Tutorial will show you how.

Previous: Namespaced Keywords   Up: Index   Next: Talking To Servers

Talking To Servers

This page describes how a re-frame app might "talk" to a backend HTTP server.

We'll assume there's a json-returning server endpoint at "http://json.my-endpoint.com/blah". We want to GET from that endpoint and put a processed version of the returned json into `app-db`.

Triggering The Request

The user often does something to trigger the process.

Here's a button which the user could click:

```
(defn request-it-button
  []
  [:div {:class "button-class"
         :on-click  #(dispatch [:request-it])}  ;; get data from the server !!
         "I want it, now!"])
```

Notice the `on-click` handler - it `dispatch`es the event `[:request-it]`.

The Event Handler

That `:request-it` event will need to be "handled", which means an event handler must be registered for it.

We want this handler to:

1. Initiate the HTTP GET
2. Update a flag in `app-db` which will trigger a modal "Loading ..." message for the user to see

We're going to create two versions of this event handler. First, we'll create a problematic version of the event handler and then, realising our sins, we'll write a second version which is a soaring paragon of virtue. Both versions will teach us something.

Version 1

We're going to use the cljs-ajax library as the HTTP workhorse.

Here's the event handler:

```
(ns my.app.events                   ;; <1>
   (:require [ajax.core :refer [GET]]
             [re-frame.core :refer [reg-event-db]))

(reg-event-db        ;; <-- register an event handler
  :request-it        ;; <-- the event id
  (fn                ;; <-- the handler function
    [db _]

    ;; kick off the GET, making sure to supply a callback for success and failure
    (GET
      "http://json.my-endpoint.com/blah"
      {:handler       #(dispatch [:process-response %1])   ;; <2> further dispatch !!
       :error-handler #(dispatch [:bad-response %1])})     ;; <2> further dispatch !!

     ;; update a flag in `app-db` ... presumably to cause a "Loading..." UI
     (assoc db :loading? true)))    ;; <3> return an updated db
```

Further Notes:

1. Event handlers are normally put into an `events.cljs` namespace
2. Notice that the GET callbacks issue a further `dispatch`. Such callbacks should never attempt to close over `db` themselves, or make any changes to it because, by the time these callbacks happen, the value in `app-db` may have changed. Whereas, if they `dispatch`, then the event handlers looking after the event they dispatch will be given the latest copy of the db.
3. event handlers registered using `reg-event-db` must return a new value for `app-db`. In our case, we set a flag which will presumably cause a "Loading ..." UI to show.

Successful GET

As we noted above, the on-success handler itself is just `(dispatch [:process-response RESPONSE])`. So we'll need to register a handler for this event too.

Like this:

```
(reg-event-db
  :process-response
  (fn
    [db [_ response]]           ;; destructure the response from the event vector
    (-> db
        (assoc :loading? false) ;; take away that "Loading ..." UI
        (assoc :data (js->clj response))))  ;; fairly lame processing
```

A normal handler would have more complex processing of the response. But we're just sketching here, so we've left it easy.

There'd also need to be a handler for the `:bad-response` event too. Left as an exercise.

Problems In Paradise?

This approach will work, and it is useful to take time to understand why it would work, but it has a problem: the event handler isn't pure.

That `GET` is a side effect, and side effecting functions are like a well salted paper cut. We try hard to avoid them.

Version 2

The better solution is, of course, to use an effectful handler. This is explained in detail in the previous tutorials: Effectful Handlers and Effects.

In the 2nd version, we use the alternative registration function, `reg-event-fx`, and we'll use an "Effect Handler" supplied by this library https://github.com/Day8/re-frame-http-fx. You may soon feel confident enough to write your own.

Here's our rewrite:

```
(ns my.app.events
   (:require
      [ajax.core :as ajax]
      [day8.re-frame.http-fx]
      [re-frame.core :refer [reg-event-fx]]))

(reg-event-fx        ;; <-- note the `-fx` extension
  :request-it        ;; <-- the event id
  (fn                ;; <-- the handler function
    [{db :db} _]     ;; <-- 1st argument is coeffect, from which we extract db

    ;; we return a map of (side) effects
    {:http-xhrio {:method          :get
                  :uri             "http://json.my-endpoint.com/blah"
                  :format          (ajax/json-request-format)
                  :response-format (ajax/json-response-format {:keywords? true})
                  :on-success      [:process-response]
                  :on-failure      [:bad-response]}
     :db  (assoc db :loading? true)}))
```

Notes:

1. Our event handler "describes" side effects, it does not "do" side effects
2. The event handler we wrote for `:process-response` stays as it was

Previous: Loading Initial Data   Up: Index   Next: Subscribing to External Data

Table Of Contents

Subscribing to External Data

In Talking To Servers we learned how to communicate with servers using both pure and effectful handlers. This is great, but what if you want to query external data using subscriptions the same way you query data stored in `app-db` ? This tutorial will show you how.

There Can Be Only One!!

`re-frame` apps have a single source of data called `app-db` .

The `re-frame` README asks you to imagine `app-db` as something of an in-memory database. You query it (via subscriptions) and transactionally update it (via event handlers).

Components Don't Know, Don't Care

Components never know the structure of your `app-db` , much less its existence.

Instead, they `subscribe` , declaratively, to data, like this `(subscribe [:something "blah"])` , and that allows Components to obtain a stream of updates to "something", while knowing nothing about the source of the data.

A 2nd Source

All good but ... SPAs are seldom completely self contained data-wise.

There's a continuum between apps which are 100% standalone data-wise, and those where remote data is utterly central to the app's function. In this page, we're exploring the remote-data-centric end of this continuum.

And just to be clear, when I'm talking about remote data, I'm thinking of data luxuriating in remote databases like firebase, rethinkdb, PostgreSQL, Datomic, etc

- data sources that an app must query and mutate.

So, the question is: how would we integrate this kind of remote data into an app when re-frame seems to have only one source of data: `app-db` ?
How do we introduce a second or even third source of data? How should we `subscribe` to this remote data, and how would we `update` it?

By way of explanation, let's make the question specific: how could we wire up a Component which displays a collection of `items` , when those items come from a remote database?

In your mind's eye, imagine this kind of query against that remote database: `select id, price, description from items where type="see through"` .

Via A Subscription

In `re-frame` , Components always obtain data via a subscription. Always.

So, our Component which shows items is going to

```
(let [items (re-frame/subscribe [:items "see through"]) ...
```

and the subscription handler will deliver them.

Which, in turn, means our code must have a subscription handler defined:

```
(re-frame/reg-sub
  :items
  (fn [db [_ item-type]
    ...))
```

Which is fine ... except we haven't really solved this problem yet, have we?
We've just transferred the problem away from the Component and into the subscription handler?

Well, yes, we have, and isn't that a fine thing!! That's precisely what we want from our subscription handlers ... to manage how the data is sourced ... to hide that from the Component.

The Subscription Handler's Job

Right, so let's write the subscription handler.

There'll be code in a minute but, first, let's describe how the subscription handler will work:

1. Upon being required to provide items, it has to issue a query to the remote database. Perhaps this will be done via a RESTful GET. Or via a firebase connection. Or by pushing a JSON representation of the query down a websocket. Something. And it is the subscription handler's job to know how it is done.
2. This query be async - with the results arriving sometime "later". And when they eventually arrive, the handler must organise for the query results to be placed into `app-db` , at some known, particular path. In the meantime, the handler might want to ensure that the absence of results is also communicated to the Component, allowing it to display "Loading ...". The Nine States of Design has some useful information on designing your application for different states that your data might be in.
3. The subscription handler must return something to the Component. It should give back a `reaction` to that known, particular path within `app-db` , so that when the query results eventually arrive, they will flow through into the Component for display.
4. The subscription handler will detect when the Component is destroyed and no longer requires the subscription. It will then clean up, getting rid of those now-unneeded items, and sorting out any stateful database connection issues.

Notice what's happening here. In many respects, `app-db` is still acting as the single source of data. The subscription handler is organising for the right remote data to "flow" into `app-db` at a known, particular path, when it is needed by a Component. And, equally, for this data to be cleaned up when it is no longer required.

Some Code

Enough fluffing about with words, here's a code sketch for our subscription handler:

```
(re-frame/reg-sub-raw
  :items
  (fn [app-db [_ type]]
       (let  [query-token (issue-items-query!
                            type
```

```
                             :on-success #(re-frame/dispatch [:write-to  [:some :path]]))]
           (reagent.ratom/make-reaction
             (fn [] (get-in @app-db [:some :path] []))
             :on-dispose #(do (terminate-items-query! query-token)
                              (re-frame/dispatch [:cleanup [:some :path]]))))))
```

A few things to notice:

1. We are using the low level `reg-sub-raw` to register our handler (and not the more normal `reg-sub`) so we can get an `:on-dispose` callback when the subscription is no longer needed. See the `reg-sub-raw` docs at the end of this tutorial
2. You have to write `issue-items-query!`. Are you making a Restful GET? Are you writing JSON packets down a websocket? The query has to be made.
3. We do not issue the query via a `dispatch` because, to me, it isn't an event. But we most certainly do handle the arrival of query results via a `dispatch` and associated event handler. That to me is an external event happening to the system. The event handler can curate the arriving data in whatever way makes sense. Maybe it does nothing more than to `assoc` into an `app-db` path, or maybe this is a rethinkdb changefeed subscription and your event handler will have to collate the newly arriving data with what has previously been returned. Do what needs to be done in that event handler, so that the right data will be put into the right path.
4. We use Reagent's `make-reaction` function to create a reaction which will return that known, particular path within `app-db` where the query results are to be placed.
5. We use the `on-dispose` callback on this reaction to do any cleanup work when the subscription is no longer needed. Clean up `app-db`? Clean up the database connection?

Any Good?

It turns out that this is a surprisingly flexible and clean approach. And pretty damn obvious once someone points it out to you (which is a good sign). There's a lot to like about it.

For example, if you are using rethinkdb, which supports queries which yield "change feeds" over time, rather than a one-off query result, you have to actively close such queries when they are no longer needed. That's easy to do in our cleanup code.

We can source some data from both PostgreSQL and firebase in the one app, using the same pattern. All remote data access is done in the same way.

Because query results are `dispatched` to an event handler, you have a lot of flexibility about how you process them.

The whole set of pieces can be arranged and tweaked in many ways. For example, with a bit of work, we could keep a register of all currently used queries. And then, if ever we noticed that the app had gone offline, and then back online, we could organise to reissue all the queries again (with results flowing back into the same known paths), avoiding stale results.

Also, notice that putting ALL interesting data into `app-db` has nice flow on effects. In particular, it means it is available to event handlers, should they need it when servicing events (event handlers get `db` as a parameter, right?).
If this item data was held in a separate place, other than `app-db`, it wouldn't be available in this useful way.

Warning: Undo/Redo

This technique caches remote data in `app-db`. Be sure to exclude this cache area from any undo/redo operations using the available configuration options

Query De-duplication

In v0.8.0 of re-frame onwards, subscriptions are automatically de-duplicated.

In prior versions, in cases where the same query is simultaneously issued from multiple places, you'd want to de-duplicate the queries. One possibility is to do this duplication in `issue-items-query!` itself. You can `count` the duplicate queries and only clear the data when that count goes to 0.

Thanks To

@nidu for his valuable review comments and insights

The Alternative Approach

Event handlers do most of the heavy lifting within re-frame apps.

When buttons get clicked, or items get dragged 'n dropped, or tabs get chosen, they know how to transition the app from one state to the next. That's their job. And, when they make such a transition, it is quite reasonable to expect them to ALSO source the data needed in the new state.

So there's definitely a case for NOT using the approach outlined above and, instead, making event handlers source data and plonk it into a certain part of `app-db` for use by subscriptions.

In effect, there's definitely an argument that subscriptions should only ever source from `app-db` BUT that it is event handlers which start and stop the sourcing of data from remote places.

Sorry, but you'll have to work out which of these two variations works best for you.

Within this document the first alternative has been given more word count only because there's a few more tricks to make it work, not because it is necessarily preferred.

Absolutely Never Do This

Sometimes, because of their background with other JS frameworks, new re-framians feel like the Components themselves (the views) should have the responsibility of sourcing the data they need.

They then use React lifecycle methods like `:component-did-mount` to load remote data.

I believe this is absolutely the wrong way to do it.

In re-frame, we want views to be as simple and dumb as possible. They turn data into HTML and nothing more. they absolutely do not do imperative stuff.

Use one of the two alternatives described above.

Previous: Talking to Servers   Up: Index

Debugging Event Handlers

This page describes techniques for debugging re-frame's event handlers.

Event handlers are quite central to a re-frame app. Only event handlers can update `app-db` to "step" an application "forward" from one state to the next.

The `debug` Interceptor

You might wonder: is my event handler making the right changes to `app-db` ?

During development, the built-in `re-frame.core/debug` interceptor can help. It writes to `console.log` :

1. the event being processed, for example: `[:attempt-world-record true]`
2. the changes made to `db` by the handler in processing the event

`debug` uses `clojure.data/diff` to compare the value of `app-db` before and after the handler ran, showing what changed.

clojure.data/diff returns a triple , the first two entries of which `debug` will display in `console.log` (the 3rd says what has not changed and isn't interesting).

The output produced by `clojure.data/diff` can take some getting used to, but you should stick with it -- your effort will be rewarded.

Using `debug`

So, you will add this Interceptor like this:

```
(re-frame.core/reg-event-db
   :some-id
   [re-frame.core/debug]         ;;  <----  added here!
   some-handler-fn)
```

Except, of course, we need to be more deft - we only want `debug` in development builds. We don't want the overhead of those `clojure.data/diff` calculations in production. So, this is better:

```
(re-frame.core/reg-event-db
   :some-id
   [(when ^boolean goog.DEBUG re-frame.core/debug)]   ;;  <----  conditional!
   some-handler-fn)
```

`goog.DEBUG` is a compile time constant provided by the `Google Closure Compiler` . It will be `true` when the build within `project.clj` is `:optimization :none` and `false` otherwise.

Ha! I see a problem, you say. In production, that `when` is going to leave a `nil` in the interceptor vector. So the Interceptor vector will be `[nil]` . Surely that's a problem?

Well, actually, no it isn't. re-frame filters out any `nil` from interceptor vectors.

Too Much Repetition - Part 1

Each event handler has its own interceptor stack.

That might be all very flexible, but does that mean we have to put this `debug` business on every single handler? That would be very repetitive.

Yes, you will have to put it on each handler. And, yes, that could be repetitive, unless you take some steps.

One thing you can do is to define standard interceptors at the top of the `event.cljs` namespace:

```
(def standard-interceptors  [(when ^boolean goog.DEBUG debug)  another-interceptor])
```

And then, for any one event handler, the code would look like:

```
(re-frame.core/reg-event-db
   :some-id
   standard-interceptors        ;; <--- use the common definition
   some-handler-fn)
```

or perhaps:

```
(re-frame.core/reg-event-db
   :some-id
   [standard-interceptors specific-interceptor]  ;; mix with something specific
   some-handler-fn)
```

So that `specific-interceptor` could be something required for just this one event handler, and it can be combined the standard ones.

Wait on! "I see a problem", you say. `standard-interceptors` is a `vector`, and it is within another `vector` along side `specific-interceptor` - so that's nested vectors of interceptors!

No problem, re-frame uses `flatten` to take out all the nesting - the result is a simple chain of interceptors. And, as we have discussed, `nil`s are removed.

3. Checking DB Integrity

Always have a detailed schema for the data in `app-db`!

Why?

First, schemas serve as invaluable documentation. When I come to a new app, the first thing I want to look at is the underlying information model - the schema of the data. I hope it is well commented and I expect it to be rigorous and complete, using Clojure spec or, perhaps, a Prismatic Schema.

Second a good spec allows you to assert the integrity and correctness of the data in `app-db`. Because all the data is in one place, that means you are asserting the integrity of ALL the data in your app, at one time. All of it.

When should we do this? Ideally, every time a change is made!

Well, it turns out that only event handlers can change the value in `app-db`, so only an event handler could corrupt it. So, we'd like to **recheck the integrity of `app-db` immediately after *every* event handler has run**.

All of it, every time. This allows us to catch any errors very early, easily assigning blame (to the rouge event handler).

Schemas are typically put into `db.cljs` (see the todomvc example in the re-frame repo). Here's an example using Prismatic Schema (although a more modern choice would be to use Clojure spec):

```
(ns my.namespace.db
  (:require
    [schema.core :as s]))

;; As exactly as possible, describe the correct shape of app-db
;; Add a lot of helpful comments. This will be an important resource
;; for someone looking at you code for the first time.
(def schema
  {:a {:b s/Str
       :c s/Int}
   :d [{:e s/Keyword
        :f [s/Num]}]})
```

And a function which will check a db value against that schema:

```
(defn valid-schema?
  "validate the given db, writing any problems to console.error"
  [db]
  (let [res (s/check schema db)]
    (if (some? res)
      (.error js/console (str "schema problem: " res)))))
```

Now, let's organise for `valid-schema?` to be run **after** every handler. We'll use the built-in `after` Interceptor factory function:

```
(def standard-interceptors [(when ^boolean goog.DEBUG debug)
                            (when ^boolean goog.DEBUG (after db/valid-schema?))]) ;; <-- new
```

Now, the instant a handler messes up the structure of `app-db` you'll be alerted. But this overhead won't be there in production.

Too Much Repetition - Part 2

Above, we discussed a way of "factoring out" common interceptors into `standard-interceptors`.

There's an additional technique we can use to ensure that all event handlers get certain Interceptors: you write a custom registration function -- a replacement for `reg-event-db` -- like this:

```
(defn my-reg-event-db          ;; alternative to reg-event-db
  ([id handler-fn]
    (re-frame.core/reg-event-db id standard-interceptors handler-fn))
  ([id interceptors handler-fn]
    (re-frame.core/reg-event-db
        id
        [standard-interceptors interceptors]
        handler-fn)))
```

Notice how this registration function inserts our standard interceptors every time.

From now on, you can register your event handlers like this and know that the two standard Interceptors have been inserted:

```
(my-reg-event-db      ;; <-- adds std interceptors automatically
  :some-id
  some-handler-fn)
```

What about the -fx variation?

Above we created `my-reg-event-db` as a new registration function for `-db` handlers. Now, `-db` handlers take `db` and `event` arguments, and return a new `db`. So, they MUST return a new `db` value.

But what if we tried to do the same for `-fx` handlers which, instead, return an `effects` map which may, or may not, contain a `:db`? Our solution would have to allow for the absence of a new `db` value (by doing no validity check, because nothing was being changed).

```
(def debug? ^boolean goog.DEBUG)
(def standard-interceptors-fx
  [(when debug?  debug)    ;; as before
   (when debug? (after #(if % (db/valid-schema? %)))]) ;; <-- different after
```

and then:

```
(defn my-reg-event-fx          ;; alternative to reg-event-db
  ([id handler-fn]
    (re-frame.core/reg-event-fx id standard-interceptors-fx handler-fn))
  ([id interceptors handler-fn]
    (re-frame.core/reg-event-fx
        id
        [standard-interceptors-fx interceptors]
        handler-fn)))
```

Debugging

This page describes a technique for debugging re-frame apps. It proposes a particular combination of tools.

Know The Beast!

re-frame apps are **event driven**.

Event driven apps have this core, perpetual loop:

1. your app is in some quiescent state, patiently waiting for the next event
2. an event arrives (because the user presses a button, a websocket gets data, etc)
3. computation/processing follows as the event is handled, leading to changes in app state, the UI, etc
4. Goto 1

When debugging an event driven system, our focus will be step 3.

re-frame's Step 3

With re-frame, step 3 happens like a **domino sequence**: an event arrives and then bang, bang, bang, one domino triggers the next:

- Event dispatch
- Event handling
- Effects handling
- subscription handlers
- view functions

Every single event is processed in the same way. Every single one. A delightfully regular environment to understand and debug!

Observe The Beast

Bret Victor has explained to us the importance of **observability**. In which case, when we are debugging re-frame, what do we want to observe?

re-frame's domino process involves *data values flowing in and out of relatively simple, pure functions*. Derived data flowing. So, to debug we want to observe:

- which functions are called
- what data flowed in and out of them

Functions and data: What data was in the event? What event handler was then called? What interceptors then ran? What state changes did that event handler cause? What subscription handlers were then triggered? What new values did they then return? And which Reagent components then rerendered? What hiccup did they return? It's all just functions processing data.

So, in Clojurescript, how do we observe functions and data? Well, as luck would have it, ClojureScript is a lisp and it is readily **traceable**.

How To Trace?

Below, I suggest a particular combination of technologies which, working together, will write a trace to the devtools console. Sorry, but there's no fancy SVG dashboard. We said simple, right?

First, use `clairvoyant` to trace function calls and data flow. We've had a couple of Clairvoyant PRs accepted, and they make it work well for us. We've also written a specific Clairvoyant tracer tuned for our re-frame needs. https://clojars.org/day8/re-frame-tracer.

Second, use cljs-devtools because it allows you to inspect traced data. ~~That means you'll need to be using a very fresh version of Chrome. But it is worth it.~~

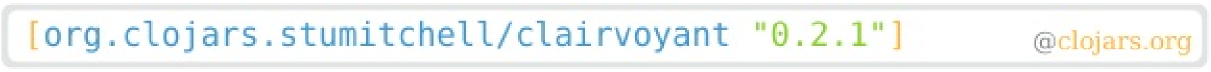
Finally, because we want you to easily scan, parse and drill into trace data, we'll be using Chrome devtool's `console.group()` and `console.endGroup()`.

Your browser

You'll need to install `clj-devtools` by following these instructions.

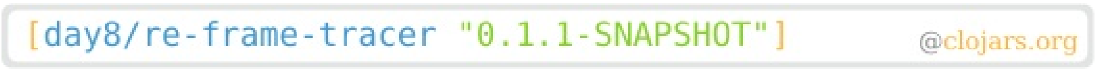
Your Project

Add these to your project.clj `:dependencies`. First up a private fork of clairvoyant.



[org.clojars.stumitchell/clairvoyant "0.2.1"] @clojars.org

Then the customised tracer for cljs-devtools that includes a colour choice



[day8/re-frame-tracer "0.1.1-SNAPSHOT"] @clojars.org

Next, we're going to assume that you have structured you app in the recommended way, meaning you have the namespaces `events.cljs`, `subs.cljs` and `views.cljs`. It is the functions within these namespaces that we wish to trace.

1. At the top of each add these namespaces, add these requires:

   ```
   [clairvoyant.core :refer-macros [trace-forms]]
   [re-frame-tracer.core :refer [tracer]]
   ```

2. Then, immediately after the `ns` form add (if you want a green colour):

   ```
   (trace-forms {:tracer (tracer :color "green")}
   ```

3. Finally, put in a closing `)` at the end of the file. Now all functions within the `ns` will be traced. It that is too noisy -- perhaps you won't want to trace all the helper functions -- then you can move the wrapping macros `trace-froms` around to suit your needs.

4. Colour choice

   We have sauntered in the direction of the following colours

   | file | colour| |--------------|-------| | `handlers.clj` | green | | `subs.cljs` | brown | | `views.clj` | gold |

   But I still think orange, flared pants are a good look. So, yeah. You may end up choosing others.

Say No To Anonymous

To get good quality tracing, you need to provide names for all your functions. So, don't let handlers be anonymous when registering them.

For example, make sure you name the renderer in a Form2 component:

```
(defn my-view
  []
  (let [name   (subscribe [:name])]
```

```
    (fn my-view-renderer []                ;;   <--  name it!!
      [:div @name])))
```

And name those event handlers:

```
(reg-event-db
  :blah
  [interceptors]
  (fn blah-handler    ;;   <-- name it
    [db v]
    (assoc db :blah true)))
```

IMPORTANT

By default, our clairvoyant fork does not produce any trace!!

You must throw a compile-time switch for tracing to be included into development builds.

If you are using `lein`, do this in your `project.clj` file:

```
:cljsbuild {:builds [{:id "dev"            ;; for the development build, turn on tracing
                      ....
                      :compiler {
                          :closure-defines {"clairvoyant.core.devmode" true}
                      }}]}
```

So, just to be clear, if you see no tracing when you are debugging, it is almost certainly because you haven't successfully turned on this switch. Your production builds need to nothing because, by default, all trace is compiled out of the code.

The result

Load your app, and open the dev-tools console. Make an event happen (click a button?). Notice the colour coded tracing showing the functions being called and the derived data flowing.

Do you see the dominos?

Warning

If the functions you are tracing take large data-structures as parameters, or return large values, then you will be asking clairvoyant to push/log a LOT of data into the `js/console`. This can take a while and might mean devtools takes a lot of RAM.

For example, if your `app-db` was big and complicated, you might use `path` middleware to "narrow" that part of `app-db` passed into your event handler because logging all of `app-db` to `js/console` might take a while (and not be that useful).

React Native

If you have not enabled Remote JS Debugging in the emulator you will get the following error related to console.groupCollapsed:

```
[TypeError: console.groupCollapsed is not a function. (In 'console.groupCollapsed("%c%s",[cljs.core.str("color:"),clj
s.core.str(self__.color),cljs.core.str(";")].join(''),title)', 'console.groupCollapsed' is undefined)] line: 112, col
umn: 23
```

Enable **Debug JS Remotely** to fix this.

Appendix A - Prior to V0.8.0

If you are using v0.8.0 or later, then you can ignore this section.

Prior to v0.8.0, subscriptions were done using `re-frame.core/reg-sub-raw` , instead of `re-frame.core/reg-sub` (which is now the preferred method).

Details of the changes can be found here.

When using `re-frame.core/reg-sub-raw` , you must explicitly use `reaction` . And unfortunately both `trace-forms` and `reaction` are macros and they don't work well together. So there is some necessary changes to your `reg-sub-raw` code to get them to work with clairvoyant, you need to replace the macro `reaction` with the function `make-reaction` .

Do the following code:

```
(ns my.ns
 (:require-macros [reagent.ratom :refer [reaction]]))

;; ...

(re-frame.core/reg-sub-raw
 :my-sub
 (fn
   [db _]
   (reaction (get-in @db [db-root :my-sub]))))
```

needs to become

```
(ns my.ns
 (:require [reagent.ratom :refer [make-reaction]]))

;; ...

(subs/register
 :my-sub
 (fn
   [db _]
   (make-reaction (fn my-subscription
                    []
                   (get-in @db [db-root :my-sub])))))
```

From @mccraigmccraig we get the following (untested by me, but they look great):

> I finally had enough of all the boilerplate required to use clairvoyant with re-frame subs & handlers and wrote some code to tidy it up...

```
(ns er-webui.re-frame
  (:require
   [clojure.string :as str]
   [clojure.pprint :as pp]
   [clairvoyant.core]
   [cljs.analyzer :as analyzer]))

(def expand-macros
  #{`reaction
    `regsub
    `reghandler})

(defn expand-op?
  "should the op represented by the sym be expanded...
   expands the sym to its fully namespaced version and
   checks against expand-macros"
  [sym env]
  (when-let [{var-name :name} (analyzer/resolve-macro-var env sym)]
    ;; (pp/pprint ["expand-op?" sym var-name] *err*)
    (expand-macros var-name)))

(defn maybe-expand
  "recursively descend forms calling macroexpand-1
   on any forms with a symbol from expand-macros in
   first position"
  [form env]
  (if (and (seq? form)
           (symbol? (first form)))
    (let [[op & r] form
          resolved-op (expand-op? op env)]
      (if resolved-op
        (maybe-expand
         (macroexpand-1 (cons resolved-op r))
         env)
        (cons op
              (doall (for [f r]
                       (maybe-expand
                        f
                        env))))))
    form))

(defn maybe-expand-forms
  [forms env]
  (doall
   (for [form forms]
     (let [exp (maybe-expand form env)]
       (when (not= exp form)
         ;; (pp/pprint exp *err*)
         )
       exp))))

(defn fn-name
  "make a sensible fn name from
   a possibly namespaced symbol or keyword"
  ([k] (fn-name k ""))
  ([k suffix]
   (assert (or (keyword? k) (symbol? k)))
   (-> k
       (str suffix)
       (str/replace #"^:" "")
       (str/replace #"\." "-")
       (str/replace "/" "--")
       symbol)))

(defmacro reaction
  "like reagent.core/reaction except it gives the fn a name
   which makes for useful tracing"
  [reaction-name & body]
  (let [reaction-fn-name# (fn-name reaction-name)]
    `(reagent.ratom/make-reaction
      (~'fn ~reaction-fn-name#
       []
       ~@body))))

(defmacro regsub
  "like re-frame.core/register-sub except it creates
   the fn with a name for better tracing"
  [sub-key params & body]
  (assert (vector? params))
  (let [sub-fn-name# (fn-name sub-key)]
    `(re-frame.core/register-sub
      ~sub-key
      (~'fn ~sub-fn-name#
       ~params
       ~@body))))

(defmacro reghandler
  "like re-frame.core/register-handler except it
   creates an fn with a name which makes for better tracing"
  [handler-key middleware-or-params & body]
  (let [handler-fn-name (fn-name handler-key "-h")
        middleware (when (and (not (vector? middleware-or-params))
                              (vector? (first body)))
                     middleware-or-params)
        params (if middleware
                 (first body)
                 middleware-or-params)
        body (if middleware
               (rest body)
               body)]
    (assert (vector? params))
    `(re-frame.core/register-handler
      ~handler-key
      ~middleware
      (~'fn ~handler-fn-name
        ~params
        ~@body))))

(defmacro trace-subs
  [& body]
  (let [body-forms# (maybe-expand-forms body &env)]
    `(clairvoyant.core/trace-forms
      {:tracer (re-frame-tracer.core/tracer :color "brown")}

      ~@body-forms#)))

(defmacro trace-handlers
  [& body]
  (let [body-forms# (maybe-expand-forms body &env)]
    `(clairvoyant.core/trace-forms
      {:tracer (re-frame-tracer.core/tracer :color "blue")}

      ~@body-forms#)))

(defmacro trace-views
  [& body]
  (let [body-forms# (maybe-expand-forms body &env)]
    `(clairvoyant.core/trace-forms
      {:tracer (re-frame-tracer.core/tracer :color "green")}

      ~@body-forms#)))
```

> gives you subs like this -

```
(regsub :initialised
  [db _]
  (reaction initialised-r
    (get-in @db [:initialised])))
```

> and handlers like this -

```
(reghandler
 :after-init
  er-middleware
  [db [_]]
  (code-push/sync)
  db)
```

Testing

This is an introduction to testing re-frame apps. It walks you through some choices.

What To Test

For any re-frame app, there's three things to test:

- **Event Handlers** - most of your testing focus will be here because this is where most of the logic lives
- **Subscription Handlers** - often not a lot to test here. Only Layer 2 subscriptions need testing.
- **View functions** - I don't tend to write tests for views. There, I said it. Hey! It is mean to look at someone with that level of disapproval, while shaking your head. I have my reasons ...
  In my experience with the re-frame architecture, View Functions tend to be an unlikely source of bugs. And every line of code you write is like a ball & chain you must forevermore drag about, so I dislike maintaining tests which don't deliver good bang for buck.

And, yes, in theory there's also `Effect Handlers` (Domino 3) to test, but you'll hardly ever write one, and, anyway, each one is different, so I've got no good general insight to offer you for them. They will be ignored in this tutorial.

Test Terminology

Let's establish some terminology to aid the further explanations in this tutorial. Every unittest has 3 steps:

1. **setup** initial conditions
2. **execute** the thing-under-test
3. **verify** that the thing-under-test did the right thing

Exposing Event Handlers For Test

Event Handlers are pure functions which should make them easy to test, right?

First, create a named event handler using `defn` like this:

```
(defn select-triangle
  [db [_ triangle-id]
  ... return a modified version of db)
```

You'd register this handler in a separate step:

```
(re-frame.core/reg-event-db     ;; this is a "-db" event handler, not "-fx"
  :select-triangle
  [some-interceptors]
  select-triangle)    ;; <--- defn above. don't use an annonomous fn
```

This arrangement means the event handler function `select-triangle` is readily available to be unittested.

Event Handlers - Setup - Part 1

To test `select-triangle` , a unittest must pass in values for the two arguments `db` and `v` . And, so, our **setup** would have to construct both values.

But how to create a useful `db` value?

`db` is a map of a certain structure, so one way would be to simply `assoc` values into a map at certain paths to simulate a real-world `db` value or, even easier, just use a map literal, like this:

```
;; a test
(let [
      ;; setup - create db and event
      db      {:some 42  :thing "hello"}   ; a literal
      event   [:select-triange :other :event :args]

      ;; execute
      result-db (select-triange db event)]

      ;; validate that result-db is correct)
      (is ...)
```

This certainly works in theory, but in practice, unless we are careful, constructing the `db` value in **setup** could:

- be manual and time consuming
- tie tests to the internal structure of `app-db`

The **setup** of every test could end up relying on the internal structure of `app-db` and any change in that structure (which is inevitable over time) would result in a lot re-work in the tests. That's too fragile.

So, this approach doesn't quite work.

Event Handlers - Setup - Part 2

> In re-frame, `Events` are central. They are the "language of the system". They provide the eloquence.

The `db` value (stored in `app-db`) is the cumulative result of many event handlers running.

We can use this idea. In **setup**, instead of manually trying to create that `db` value, we could "build up" a `db` value by threading `db` through many event handlers which cumulatively create the required initial state. Tests then need know nothing about the internal structure of that `db` .

Like this:

```
(let [
      ;; setup - cummulatively build up db
      db (-> {}    ;; empty db
             (initialise-db [:initialise-db])   ;; each event handler expects db and event
             (clear-panel   [:clear-panel])
             (draw-triangle [:draw-triangle 1 2 3]))

      event  [:select-triange :other :stuff]

      ;; now execute the event handler under test
      db'    (select-triange db event)]

      ;; validate that db' is correct
      (is ...)
```

This approach works so long as all the event handlers are of the `-db` kind, but the threading gets a little messy when some event handlers are of the `-fx` kind which take a `coeffect` argument and return `effects` , instead of a `db` value.

So, this approach is quite workable in some cases, but can get messy in the general case.

Event Handlers - Setup - Part 3

There is further variation which is quite general but not as pure.

During test **setup** we could literally just `dispatch` the events which would put `app-db` into the right state.

Except, we'd have to use `dispatch-sysnc` rather `dispatch` to force immediate handling of events, rather than queuing.

```
  ;; setup - cummulatively build up db
  (dispatch-sync [:initialise-db])
  (dispatch-sync [:clear-panel])
  (dispatch-sync [:draw-triangle 1 2 3]))

  ;; execute
  (dispatch-sync  [:select-triange :other :stuff])

  ;; validate that the valuein `app-db` is correct
  ;; perhaps with subscriptions
```

Notes:

1. we use `dispatch-sync` because `dispatch` is async (event is handled not now, but soon)
2. Not pure. We are choosing to mutate the global `app-db`. But having said that, there's something about this approach with is remarkably pragmatic.
3. the **setup** is now very natural. The associated handlers can be either `-db` or `-fx`
4. if the handlers have effects other than just updating app-db, we might need to stub out XXX
5. How do we look at the results ????

If this method appeals to you, you should ABSOLUTELY review the utilities in this helper library: re-frame-test.

In summary, event handlers should be easy to test because they are pure functions. The interesting part is the unittest "setup" where we need to establishing an initial value for `db`.

Subscription Handlers

Here's a Subscription Handler from the todomvc example:

```
(reg-sub
  :visible-todos

  ;; signal function
  (fn [query-v _]
    [(subscribe [:todos])
     (subscribe [:showing])])

  ;; computation function
  (fn [[todos showing] _]   ;; that 1st parameter is a 2-vector of values
    (let [filter-fn (case showing
                      :active (complement :done)
                      :done   :done
                      :all    identity)]
      (filter filter-fn todos))))
```

How do we test this?

First, we could split the computation function from its registration, like this:

```
(defn visible-todos
  [[todos showing] _]

  (let [filter-fn (case showing
                    :active (complement :done)
                    :done   :done
                    :all    identity)]
   (filter filter-fn todos)))

(reg-sub
  :visible-todos
  (fn [query-v _]
      [(subscribe [:todos])
       (subscribe [:showing])])
  visible-todos)     ;; <--- computation function used here
```

That makes `visible-todos` available for direct unit testing. But, as we experienced with Event Handlers, the challenge is around constructing `db` values (first parameter) in a way which doesn't become fragile.

View Functions - Part 1

Components/views are more tricky and there are a few options.

But remember my ugly secret - I don't tend to write tests for my views.

But here's how, theoretically, I'd write tests if I wasn't me ...

If a View Function is Form-1, then it is fairly easy to test.

A trivial example:

```
(defn greet
   [name]
   [:div "Hello " name])

(greet "Wiki")
;;=> [:div "Hello " "Wiki"]
```

So, here, testing involves passing values into the function and checking the data structure returned for correctness.

What's returned is hiccup, of course. So how do you test hiccup for correctness?

hiccup is just a clojure data structure - vectors containing keywords, and maps, and other vectors, etc. Perhaps you'd use https://github.com/nathanmarz/specter to declaratively check on the presence of certain values and structures? Or do it more manually.

View Functions - Part 2A

But what if the View Function has a subscription?

```
(defn my-view
   [something]
   (let [val  (subscribe [:query-id])]     <-- reactive subscription
     [:div .... using @val in here])))
```

The use of `subscribe` makes the function impure (it obtains data from places other than its args).

A testing plan might be:

1. setup `app-db` with some values in the right places (via dispatch of events?)
2. call `my-view` (with a parameter) which will return hiccup
3. check the hiccup structure for correctness.

Continuing on, in a second phase you could then:

1. change the value in `app-db` (which will cause the subscription to fire)
2. call view functions again (hiccup returned).
3. check the new hiccup for correctness

Which is all possible, if a little messy.

View Functions - Part 2B

There is a pragmatic method available to handle the impurity: use `with-redefs` to stub out `subscribe`. Like this:

```
(defn subscription-stub [x]
  (atom
    (case x
      [:query-id] 42)))

(deftest some-test
  (with-redefs [re-frame/subscribe (subscription-stub)]
    (testing "some some view which does a subscribe"
      ..... call the view function and the hiccup output)))
```

For more integration level testing, you can use `with-mounted-component` from the reagent-template to render the component in the browser and validate the generated DOM.

View Functions - Part 2C

Or ... there is another option: you can structure in the first place for pure view functions.

The trick here is to create an outer and inner component. The outer sources the data (via a subscription), and passes it onto the inner as props (parameters).

As a result, the inner component, which does the testable work, is pure and easily tested. The outer is impure but trivial.

To get a more concrete idea, I'll direct you to another page in the re-frame docs which has nothing to do with testing, but it does use this `simple-outer-subscribe-with-complicated-inner-render` pattern for a different purpose: Using Stateful JS Components

Note: this technique could be made simple and almost invisible via the use of macros.

This pattern has been independently discovered by many. For example, here it is called the Container/Component pattern.

Summary

Event handlers will be your primary focus when testing. Remember to review the utilities in re-frame-test.

Frequently Asked Questions

Want To Add An FAQ?

We'd like that. Please supply a PR. Or just open an issue. Many Thanks!!

External Resources

Please add to this list by submitting a pull request.

Templates

- re-frame-template - Generates the client side SPA
- Luminus - Generates SPA plus server side.
- re-natal - React Native apps.
- Slush-reframe - A scaffolding generator for re-frame run using NodeJS. Based on re-frame `0.7.0`
- Celibidache - An opinionated starter for re-frame applications using Boot. Based on re-frame `0.7.0`

Examples and Applications Using re-frame

- BlueGenes - searching and analysing genomic data, by the University of Cambridge
- Memento a private note-taking app. Uses compojure-api, PostgreSQL and token auth.
- How to create decentralised apps with re-frame and Ethereum - Tutorial with links to code and live example.
- Braid - A new approach to group chat, designed around conversations and tags instead of rooms.
- Elfeed-cljsrn - A mobile client for Elfeed rss reader, built with React Native.
- Memory Hole - A small issue tracking app written with Luminus and re-frame.
- Crossed - A multiplayer crossword puzzle generator. Based on re-frame `0.7.0`
- imperimetric - Webapp for converting texts with some system of measurement to another, such as imperial to metric.
- Brave Clojure Open Source A site using re-frame, liberator, boot and more to display active github projects that powers http://open-source.braveclojure.com. Based on re-frame `0.6.0`
- flux-challenge with re-frame - "a frontend challenge to test UI architectures and solutions". re-frame `0.5.0`
- fractalify - An entertainment and educational webapp for creating & sharing fractal images that powers fractalify.com. Based on re-frame `0.4.1`
- boodle - A simple SPA for accounting. It uses, among others, re-frame, http-kit, compojure-api and it runs on PostgreSQL.

Effect and CoEffect Handlers

- async-flow-fx - manage a boot process dominated by async
- http-fx - performing Ajax tasks (via cljs-ajax)
- re-frame-forward-events-fx - slightly exotic
- cookie-fx - set and get cookies
- document-fx - set and get on `js/document` attributes
- re-frame-youtube-fx - YouTube iframe API wrapper
- re-frame-web3-fx - Ethereum Web3 API
- re-frame-google-analytics-fx - Google Analytics API
- re-frame-storage - Local Storage based persistence
- re-frame-storage-fx - Another take on Local Storage persistence
- re-frame-firebase - Firebase DB API

Routing

- Bidirectional using Silk and Pushy

Tools, Techniques & Libraries

- re-frame-undo - An undo library for re-frame
- re-frame-test - Advanced testing utilities
- Animation using `react-flip-move`

- re-frisk - A library for visualizing re-frame data and events.
- re-thread - A library for running re-frame applications in Web Workers.
- re-frame-datatable - DataTable UI component built for use with re-frame.
- Stately: State Machines also https://www.youtube.com/watch?v=klqorRUPluw
- re-learn - Data driven tutorials for educating users of your reagent / re-frame app

Videos

- re-frame your ClojureScript applications - re-frame presentation given at Clojure/Conj 2016
- A Video Tour of the Source Code of Ninja Tools

Server Side Rendering

- Prerenderer - Server pre-rendering library using NodeJS that works with re-frame `0.6.0` (later versions untested) Rationale Part 1 Part 2 Part 3 Release Announcement

- Server Side Rendering with re-frame - Blog post on rendering re-frame views with Clojure.

- Rendering Reagent on the Server Using Hiccup- Blog post on rendering Reagent with Clojure.

Eek! Performance Problems

Table Of Contents

1. Is It The `debug` Interceptor?

This first one is something of a non-problem.

Are you are using the `re-frame.core/debug` Interceptor? You should be, it's useful. **But** you do need to be aware of its possible performance implications.

`debug` reports what's changed after an event handler has run by using `clojure.data/diff` to do deep, CPU intensive diff on `app-db` . That diff could be taking a while, and leading to apparent performance problems.

The good news is this really isn't a production problem. `debug` should only be present in an Interceptor Chain at development time, and it should be removed from production using this technique.

Also related, anything which writes large data structures, or strings, to the js console, will be slow. So press F12, pull up devtools console, and have a good look at what's happening in there.

2. `=` On Big Structures

Reagent uses `=` to compare the previous value of a prop with the new value of that prop, when it determines if a component needs rerendering. Make sure you have a good understanding of this..

In the worst case, if those props are big data structures which differ only in some tiny, leaf aspect, then a lot of CPU cycles will be spent doing the `=` comparison only to eventually work out that, indeed, the answer is `false` .

This problem is exacerbated when components return a lot of hiccup, because lots of hiccup normally means lots of props which, in turn, means lots of `=` work to do on each of those props. Any rerender with those characteristics could end up chewing a lot of CPU cycles.

An Example Of Problem 2

Imagine you were rendering a 19 by 19 "Go" board.

And imagine that you have a high level board renderer component which creates hiccup for the 361 sub components (19 x 19 grid), and that it provides 3 props to each child:

1. grid x cord
2. grid y coord
3. a chunk of data representing the current game state, from which each of the 361 individual grid components is expected to extract the data they need to render their grid position.

This arrangement could be slow.

First, you have a parent component returning hiccup for 361 sub-components and that's a lot of hiccup!! Sure, it might not be much code - just a couple of nested `for` , but the hiccup data structure built will be substantial.

Second, after the board renderer returns all this hiccup, for every one of those 361 sub-components, Reagent must then check the 3 props to see if they are `=` to the value last rendered (to determine if they, in turn, need to be rerendered), and the comparison on the 3rd prop (game state) might be deep and expensive. Worse, we do the same expensive check 361 times in a row, and every time we get a `false` (because games state is not `=` to last time).

Third, because Reagent gets 361 `falses` , it will further rerender all 361 sub-components even though 360 of them produce the same hiccup as last time - only one position in the gird has changed.

So, when a new stone is placed on the board, and the game state changes, that triggers a large amount of unnecessary calculation, just to figure out that there's only a rendering change at one point in the 19x19 grid.

So, that's how you can get a performance problem: lots of hiccup, mixed with time consuming `=` tests on big props.

Solutions To Problem 2

The solution is to not do the unnecessary work. Duh!

Produce only the hiccup that is needed. Don't unnecessarily pass around big complicated state in props, unless you really need to.

In the Go example described above, for each new stone placed, only one point in the Go board actually needs to be rerendered, and yet our code asked Reagent to chew a lot of CPU to figure that out.

These kinds of tweaks would improve performance:

- don't give the entire game state to each of the 361 sub components and then ask them to extract what they need. Instead, give each just the state it needs, and nothing more. That will make the `=` process faster. It will also allow for Reagent to figure out that 360 of the sub components have the same props as last time, and don't need rerendering. And, so, only one sub-component will be rerendered when the parent "board level" component rerenders.
- Also, could you render the board row by row? So that less hiccup is produced by any one component? Can those rows `subscribe` to just the data for their row, so they only rerender when the row-data changes; they only generate hiccup when something really has changed?

3. Are you Using a React `key` ?

Correctly using React `keys` can also make a huge difference to performance.

Some resources:

1. http://stackoverflow.com/questions/27863039/key-property-inside-component-function
2. http://stackoverflow.com/a/37186230/5215391
3. https://groups.google.com/d/msg/reagent-project/J1ELaLV20MU/iutebA-JEgAJ

4. Callback Functions

Look at this `div` :

```
[:div  {:on-mouse-over  (fn [event] ....)  }   "hello"]
```

Every time it is rendered, that `:on-mouse-over` function will be regenerated, and it will NOT test `=` to the last time it rendered. It will appear to be a new function. It will appear to React that it has to replace the event handler.

Most of the time, this is not an issue. But if you are generating a LOT of DOM this small inefficiency can add up.

To work around the problem, lift the function out of the render. Use a Form-2 function like this:

```
(defn my-component
   []
   (let [mouse-over-cb  (fn [event] ....)  ]      ;; created once
      (fn []                                      ;; rendered many times
         [:div {:on-mouse-over  mouse-over-cb}])))
```

Now, React will see that `mouse-over-cb` is the same as last time. It won't think the event handler has been replaced.

But like I say, don't be too paranoid about this, it is unlikely to be an issue unless you have something like a table with a lot of rows.

A Weapon

Of course, the way to really track down what is going on is to use the OFFICIAL debugging technique. See the four dominoes play out in the console. You may be surprised by what you find is happening.

Be aware that tracing adds its own performance drag - there's the overhead of all that stuff getting written on the js console. Especially if the data getting traced is big - for example, tracing all of `app-db` in the console can take a while and force Chrome devtools to take masses of RAM. So you may want to selectively add tracing when poking about.

Solving The CPU Hog Problem

Sometimes a handler has a lot of CPU intensive work to do, and getting through it will take a while.

When a handler hogs the CPU, nothing else can happen. Browsers only give us one thread of execution and that CPU-hogging handler owns it, and it isn't giving it up. The UI will be frozen and there will be no processing of any other handlers (eg: `on-success` of POSTs), etc, etc. Nothing.

And a frozen UI is a problem. GUI repaints are not happening. And user interactions are not being processed.

How are we to show progress updates like "Hey, X% completed"? Or how can we handle the user clicking on that "Cancel" button trying to stop this long running process?

We need a means by which long running handlers can hand control back for "other" processing every so often, while still continuing on with their computation.

The re-frame Solution

First, all long running, CPU-hogging processes are put in event handlers. Not in subscriptions. Not in components. Not hard to do, but worth establishing as a rule, right up front.

Second, you must be able to break up that CPU work into chunks. You need a way to do part of the work, pause, then resume from where you left off (more in a minute).

In a perfect world, each chunk would take something like 16ms (60 fps). If you go longer, say 50ms or 100ms, it is no train smash, but UI responsiveness will degrade and animations, like busy spinners, will get jerky. Shorter is better, but less than 16ms delivers no added smoothness.

Third, within our handler, after it completes one unit (chunk) of work, it should not continue straight on with the next. Instead, it should do a `dispatch` to itself and, in the event vector, include something like the following:

1. a flag to say the work is not finished
2. the working state so far; and
3. what chunk to do next.

A Sketch

Here's an `-fx` handler which counts up to some number in chunks:

```
(re-frame.core/reg-event-fx
  :count-to
  (fn
    [{db :db} [_ first-time so-far finish-at]]
    (if first-time
      ;; We are at the beginning, so:
      ;;     - modify db, causing popup of Modal saying "Working ..."
      ;;     - begin iterative dispatch. Give initial version of "so-far"
      {:dispatch [:count-to false {:counter 0} finish-at]  ;; dispatch to self
       :db (assoc db :we-are-working true)}
      (if (> (:counter so-far) finish-at)
        ;; We are finished:
        ;;  - take away the state which causes the modal to be up
        ;;  - store the result of the calculation
        {:db (-> db
                 (assoc :fruits-of-labour (:counter so-far)) ;; remember the result
                 (assoc :we-are-working false))}             ;; no more modal
        ;; Still more work to do
        ;;   - run the calculation
        ;;   - redispatch, passing in new running state
```

```
          (let [new-so-far   (update so-far :counter inc)]
            {:dispatch [:count-to false new-so-far finish-at]}))))
```

Why Does A Redispatch Work?

A `dispatched` event is handled asynchronously. It is queued and not actioned straight away.

And here's the key: **After handling current events, re-frame yields control to the browser**, allowing it to render any pending DOM changes, etc. After it is finished, the browser will hand control back to the re-frame router loop, which will then handle any other queued events which, in our case, would include the event we just dispatched to perform the next chunk of work.

When the next dispatch is handled, a next chunk of work will be done, and then another `dispatch` will happen. And so on. `dispatch` after `dispatch` . Chunk after chunk. In 16ms increments if we are very careful (or some small amount of time less than, say, 100ms). But with the browser getting a look-in after each iteration.

Variations

As we go, the handler could be updating some value in `app-db` which indicates progress, and this state would then be rendered into the UI.

At a certain point, when all the work is done, the handler will likely put the fruits of its computational labour into `app-db` and clear any flags which might, for example, cause a modal dialog to be showing progress. And the process would then be done.

Cancel Button

It is a flexible pattern. For example, it can be tweaked to handle a "Cancel' button ...

If there was a “Cancel” button to be clicked, we might `(dispatch [:cancel-it])` and then have this event’s handler tweak the `app-db` by adding `:abandonment-required` flags. When a chunk-processing-handler next begins, it could check for this `:abandonment-required` flag, and, if found, stop the CPU intensive process (and clear the abandonment flags).
When the abandonment-flags are set, the UI could show "Abandoning process ..." and thus appear responsive to the user's click on “Cancel”.

That's just one approach. You can adapt the pattern as necessary.

Further Notes

Going to this trouble is completely unnecessary if the long running task involves I/O (GET, POST, HTML5 database action?) because the browser will handle I/O in another thread and give UI activities plenty of look in.

You only need to go to this trouble if it is your code which is hogging the CPU.

Forcing A One Off Render

Imagine you have a process which takes, say, 5 seconds, and chunking is just too much effort.

You lazily decide to leave the UI unresponsive for that short period.
Except, you aren't totally lazy. If there was a button which kicked off this 5 second process, and the user clicks it, you’d like the UI to show a response. Perhaps it could show a modal popup thing saying “Doing X for you”.

At this point, you still have a small problem to solve. You want the UI to show your modal message before you then hog the CPU for 5 seconds.

Updating the UI means altering `app-db` . Remember, the UI is a function of the data in `app-db` . Only changes to `app-db` cause UI changes.

So, to show that Modal, you’ll need to `assoc` some value into `app-db` and have that new value change what is rendered in your reagent components.

You might be tempted to do this:

```
(re-frame.core/reg-event-db
  :process-x
  (fn
   [db event-v]
   (assoc db :processing-X  true)    ;; hog the CPU
   (do-long-process-x)))    ;; update state, so reagent components render a modal
```

But that is just plain wrong. That `assoc` into `db` is not returned (and it must be for a `-db` handler).
And, even if that did somehow work, then you continue hogging the thread with `do-long-process-x`. There's no chance for any UI updates because the handler never gives up control. This handler owns the thread right through.

Ahhh, you think. I know what to do! I'll use that pattern I read about in the Wiki, and `re-dispatch` within an `-fx` handler:

```
(re-frame.core/reg-event-fx
  :process-x
  (fn
    [{db :db} event-v]
    {:dispatch  [:do-work-process-x]   ;; do processing later, give CPU back to browser.
     :db (assoc  db  :processing-X true)})) ;; ao the modal gets rendered

(re-frame.core/reg-event-db
  :do-work-process-x
  (fn [db _]
    (do-long-process-x db)))   ;; return a new db, presumably containing work done
```

So close. But it still won’t work. There's a little wrinkle.

That event handler for `:process-x` will indeed give back control to the browser. BUT, because of the way reagent works, that `assoc` on `db` won't trigger DOM updates until the next animation frame runs, which is 16ms away.

So, you will be yielding control to the browser, but for next 16ms there won't appear to be anything to do. And, by then, your CPU hogging code will have got control back, and will keep control for the next 5 seconds. That nice little Dialog telling you the button was clicked and action is being taken won't show.

In these kinds of cases, where you are only going to give the UI **one chance to update** (not a repeated chances every few milli seconds), then you had better be sure the DOM is fully synced.

To do this, you put meta data on the event being dispatched:

```
(re-frame.core/reg-event-fx
  :process-x
  (fn
    [{db :db} event-v]
    {:dispatch  ^:flush-dom [:do-work-process-x]   ;; <--- NOW WITH METADATA
     :db (assoc  db  :processing-X true)}))  ;; ao the modal gets rendered
```

Notice the `^:flush-dom` metadata on the event being dispatched. Use that when you want the UI to be fully updated before the event dispatch is handled.

You only need this technique when you:

1. want the DOM to be fully updated
2. because you are going to hog the CPU for a while and not give it back. One chunk of work.

If you handle via multiple chunks you don't have to do this, because you are repeatedly handing back control to the browser/UI. Its just when you are going to tie up the CPU for a one, longish chunk.

Using Stateful JS Components

You know what's good for you, and you know what's right. But it doesn't matter - the wickedness of the temptation is too much.

The JS world is brimming with shiny component baubles: D3, Google Maps, Chosen, etc.

But they are salaciously stateful and mutative. And, you, raised in a pure, functional home, with caring, immutable parents, know they are wrong. But, my, how you still yearn for the sweet thrill of that forbidden fruit.

I won't tell, if you don't. But careful plans must be made ...

The overall plan

To use a stateful js component, you'll need to write two Reagent components:

- an **outer component** responsible for sourcing data via a subscription or r/atom or cursor, etc.
- an **inner component** responsible for wrapping and manipulating the stateful JS component via lifecycle functions.

The pattern involves the outer component, which sources data, supplying this data to the inner component **via props**.

Example Using Google Maps

```
(defn gmap-inner []
  (let [gmap    (atom nil)
        options (clj->js {"zoom" 9})
        update  (fn [comp]
                  (let [{:keys [latitude longitude]} (reagent/props comp)
                        latlng (js/google.maps.LatLng. latitude longitude)]
                    (.setPosition (:marker @gmap) latlng)
                    (.panTo (:map @gmap) latlng)))]

    (reagent/create-class
      {:reagent-render (fn []
                         [:div
                          [:h4 "Map"]
                          [:div#map-canvas {:style {:height "400px"}}]])

       :component-did-mount (fn [comp]
                              (let [canvas  (.getElementById js/document "map-canvas")
                                    gm      (js/google.maps.Map. canvas options)
                                    marker  (js/google.maps.Marker. (clj->js {:map gm :title "Drone"}))]
                                (reset! gmap {:map gm :marker marker}))
                              (update comp))

       :component-did-update update
       :display-name "gmap-inner"})))



(defn gmap-outer []
  (let [pos (subscribe [:current-position])]   ;; obtain the data
    (fn []
      [gmap-inner @pos])))
```

Notes:

- `gmap-outer` obtains data via a subscription. It is quite simple - trivial almost.
- it then passes this data **as a prop** to `gmap-inner`. This inner component has the job of wrapping/managing the stateful js component (Gmap in our case above)
- when the data (delivered by the subscription) to the outer layer changes, the inner layer, `gmap-inner`, will be given a new prop - `@pos` in the case above.
- when the inner component is given new props, its entire set of lifecycle functions will be engaged.

- the renderer for the inner layer ALWAYS renders the same, minimal container hiccup for the component. Even though the `props` have changed, the same hiccup is output. So it will appear to React as if nothing changes from one render to the next. No work to be done. React/Reagent will leave the DOM untouched.
- but this inner component has other lifecycle functions and this is where the real work is done.
- for example, after the renderer is called (which ignores its props), `component-did-update` will be called. In this function, we don't ignore the props, and we use them to update/mutate the stateful JS component.
- the props passed (in this case `@pos`) in must be a map, otherwise `(reagent/props comp)` will return nil.

Pattern Discovery

This pattern has been independently discovered by many. To my knowledge, this description of the Container/Component pattern is the first time it was written up.

Code Credit

The example gmaps code above was developed by @jhchabran in this gist:
https://gist.github.com/jhchabran/e09883c3bc1b703a224d#file-2_google_map-cljs

D3 Examples

D3 (from @zachcp):

- Blog Post: http://zachcp.org/blog/2015/reagent-d3/
- Code: https://github.com/zachcp/simplecomponent
- Example: http://zachcp.github.io/simplecomponent/

A different take on using D3: https://gadfly361.github.io/gadfly-blog/2016-10-22-d3-in-reagent.html

Advanced Lifecycle Methods

If you mess around with lifecycle methods, you'll probably want to read Martin's explanations:
https://www.martinklepsch.org/posts/props-children-and-component-lifecycle-in-reagent.html

The re-frame Logo

Who

Created by the mysterious, deep thinker, known only as @martinklepsch.

Some say he appears on high value stamps in Germany and that he once punched a horse to the ground. Others say he loves recursion so much that, in his wallet, he carries a photograph of his wallet.

All we know for sure is that he wields Sketch.app like Bruce Lee wielded nunchucks.

Genesis Theories

Great, unexplained works encourage fan theories, and the re-frame logo is no exception.

One noisy group insists that @martinklepsch's design is assertively trying to `Put the 'f' back into infinity`. They have t-shirts.

Another group speculates that he created the logo as a bifarious rainbow homage to Frank Lloyd Wright's masterpiece, the Guggenheim Museum. Which is surely a classic case of premature abstraction and over engineering. Their theory, not the Guggenheim.

The infamous "Bad Touch" faction look at the logo and see the cljs logo mating noisily with re-frame's official architecture diagram. Yes, its true, their parties are completely awesome, but you will need someone to bail you out of jail later.

For the Functional Fundamentalists, a stern bunch, the logo is a flowing poststructuralist rebuttal of OO's vowel duplication and horizontal adjacency. Their alternative approach, FF, is fine, apparently, because "everyone loves a fricative".

For his part, @martinklepsch has never confirmed any theory, teasing us instead with coded clues like "Will you please stop emailing me" and "Why did you say I hit a horse?".

Assets Where?

Within this repo, look in `/images/logo/`

Open Source Code of Conduct

In order to foster an inclusive, kind, harassment-free, and cooperative community, Day8 enforces this code of conduct on our open source projects.

Summary

Harassment in code and discussion or violation of physical boundaries is completely unacceptable anywhere in Day8’s project codebases, issue trackers, chatrooms, mailing lists, meetups, and other events. Violators will be warned by the core team. Repeat violations will result in being blocked or banned by the core team at or before the 3rd violation.

In detail

Harassment includes offensive verbal comments related to gender identity, gender expression, sexual orientation, disability, physical appearance, body size, race, religion, sexual images, deliberate intimidation, stalking, sustained disruption, and unwelcome sexual attention.

Individuals asked to stop any harassing behavior are expected to comply immediately.

Maintainers are also subject to the anti-harassment policy.

If anyone engages in harassing behavior, including maintainers, we may take appropriate action, up to and including warning the offender, deletion of comments, removal from the project’s codebase and communication systems, and escalation to GitHub support.

If you are being harassed, notice that someone else is being harassed, or have any other concerns, please contact a member of the core team or email conduct@day8.com.au immediately.

We expect everyone to follow these rules anywhere in Day8's project codebases, issue trackers, chatrooms, and mailing lists.

Finally, don't forget that it is human to make mistakes! We all do. Let’s work together to help each other, resolve issues, and learn from the mistakes that we will all inevitably make from time to time.

Thanks

Thanks to the Thoughtbot Code of Conduct, CocoaPods Code of Conduct, Bundler Code of Conduct, JSConf Code of Conduct, and Contributor Covenant for inspiration and ideas.

License