HELLHOUND

HellHound Documentation

Codamic Technologies

HellHound Documentation

Under development software

This software is in Alpha stage. It’s under heavy development and reaches the Beta stage pretty soon. If you like to be part of this process please leave us a message.

Chapter 1. What is HellHound?

HellHound is a set of libraries to create simple and elegant programs based on streams. An **HellHound** application basically is a system of components which work together in order to achieve a common goal. Components form one or more data pipelines through workflows. In general systems are a great way to manage the lifecycle and data flow or your program and components are awesome for managing the state and dependencies of different pieces of your program.

1.1. Goals

The ultimate goal of **HellHound** is to create a set of libraries for creating well orginized, simple, easy to test, easy to to scale and distribute applications using Clojure and Clojurescipt around streams and data pipelines.

Another goal of this project is to create a tool which allows developers to focus on their business logic as much as possible while **HellHound** takes care of different aspects of their distributed and scalable software in an opinionated way. We want to achieve this using different component which provides these kind of tasks.

Chapter 2. Getting Started

Before continue with document please make sure the you know enough about HellHound prerequisite.

In this section we're going to build an example application to cover some the most important concepts and features of HellHound.

In order to continue with this section you need to have Leiningen setup in your box.

The final application is available at this github repository and under the `getting-started` namespace.

2.1. Before start

The aim of this tutorial is to given you a rough overview about the **HellHound** systems. In order to continue with this tutorial you should know about clojure and Leiningen.

2.2. A little bit of theory

In order to have easier time reading this tutorial, I'm going to give you a brief introduction about some of the concepts we're going to use in this tutorial. You can read more about each of these concepts in more details later in their own sections.

- **System**: It's a hashmap describing our application system. Basically how it should work. Learn More.
- **Component**: A system is create by a set of components. You can think of a component as a unix command or a function (they are more than a function though). Just like a unix command each component has a input and output. Learn More
- **Workflow**: A workflow is a graph like data structure which defines the data flow of the system. Simply describes how to pipe the output of a component to input of another component.Learn More

2.3. What do we want to build ?

In order to gain an overview of HellHound systems we're going to build a real world application together. In this example we're going to create a simple web proxy application which gets a domain name and a port number, setup a webserver to serve the content of the given domain name and serve it locally.

HellHound is library to create data pipelines. Basically data pipelines are suitable for stream processing and those use cases which involve with data streams. **But I chose the proxy use case intentionally to demonstrate the usage of HellHound for a use case which is not a case of stream processing by default.**

2.4. Tackle the problem

For this specific usage case web need to spin up a webserver which listens to a port for any HTTP URL and contructs new requests from the incoming requests with different host (the remote host) and sent them to the actual host and response back to user with the reponses fetch from the remote host.

So we're going to create a system hashmap with two component. A `webserver` component and a `loader` component. As the name suggests `webserver` component should be responsible for getting the user requests and serving responses to the user. The `loader` component should fetch the url content from the remote host. In this system we want to connect the output of `webserver` to the input of `loader` and connect the output of `loader` to input of `webserver`. So we're going to end up with a closed system.

Now let's get our feet wet.

2.5. Installation

Add **HellHound** to your dependencies as follow:

```
[codamic/hellhound "1.0.0-alpha3"]
```

`codamic/hellhound` is a meta package which automatically install serveral HellHound libraries like `codamic/hellhound.core` and `codamic/hellhound.http`. So your can only install an specific library that you need instead of using `codamic/hellhound`. For instance you might not need the `codamic/hellhound.i18n` library or `codamic/hellhound.http` so you can just install `codamic/hellhound.core`. Systems are part of `codamic/hellhound.core` artifact.

2.6. Components

Lets start with components creation. HellHound's components are just hashmaps. Simple, good old hashmaps. Any component map should contains at least three keys, `:hellhound.component/name`, `:hellhound.component/start-fn` and `:hellhound.component/stop-fn`. The value of the `:hellhound.component/name` key should be a namespaced keyword which is the component name. We use this name to refer to the component later in our system configuration. the value of `:hellhound.component/start-fn` should be function that takes two arguments. The first argument is the component hashmap itself and the second argument is another hashmap which is called context map (In order to learn more about components in details, please checkout the components section of this documentation). The main logic of the component goes into this function. It should return a component map again. The `:hellhound.component/stop-fn` function is similar to `start-fn` but accept only one argument which is the component map itself. This function is responsible for tearing down the logic of the component and do the clean up. For example, close the connection which has been opened on `start-fn` or similar stuff.

Ok, It's time to write some code. Let's start with a really basic component skeleton for the `webserver` component. Check out the following code.

Basic skeleton for a component

```
(ns getting-started.proxy.components.web
  (:require [hellhound.component :as component])) ①

(defn start-fn   ②
  [this context]
  this)

(defn stop-fn    ③
  [this]
  this)

(defn factory   ④
  []
  (component/make-component ::server start-fn stop-fn)) ⑤
```

① In order to use `make-component` function we need the `hellhound.component` ns.

② start-fn is the entry point to each component logic. Simply think of it as the place where you have to implement the logic behind your component. In this case our `start-fn` does nothing.

③ start-fn is responsible for component termination logic, mostly cleanup. For instance, closing a socket, flushing data do disk and so on.

④ Factory function is a simple function to create instances of our component with different configuration. In this case we just creates a fix component map. Nothing special.

⑤ `make-component` function is just a shortcut function to create the component map easily. We can replace it by the actual hashmap definition of the component. We passed `::server` as the first argument to this function which is the name of our component. Component's name should be namespaced keyword.

Almost all of the components that you may encounter or create, will have the same skeleton. Let's go over the basics of a component again. Each component has a name which must be a namespaced keyword, a `start-fn` with holds the logic of the component and gets two arguments, the component map itself and a context map, and should return a component map, a `stop-fn` that gets a component map and holds the termination logic. Every component has an input stream and an output stream defined in their component-map. There are more details about components which you can read in components section.

HellHound created around and shares similar ideas as unix philosophy. Components are isolated unix of execution which read data from their input, process the data and write the result to the output. System makes a data pipeline by pipeing IO of different components to one another. You'll learn more about this concept later on this documentation.

Now that we setup a very basic component and we have better understanding about component concept. let's focus on the webserver logic and create a real world webserver component.

Web server component.

```
(ns getting-started.proxy.components.web ①
  (:require [aleph.http :as http]   ②
            [hellhound.component :as hcomp]
            [manifold.stream :as stream]  ③
            [manifold.deferred :as d])) ④


(defn handler       ⑤
  [input output]
  (stream/consume #(d/success!     ⑥
                    (:response-deferred %)
                    (:response %))
                  input)
  (fn [req]               ⑦
    (let [response (d/deferred)]     ⑧
      (stream/put! output {:request req    ⑨
                           :response-deferred response})
      response)))

(defn start!   ⑩
  [port]
  (fn [this context]
    (let [[input output] (hcomp/io this)]    ⑪
      (assoc this
             :server
             (http/start-server (handler input output) {:port port})))))   ⑫

(defn stop!  ⑬
  "Stops the running webserver server."
  [this]
  (if (:server this)
    (do
      (.close (:server this))    ⑭
      (dissoc this :server))     ⑮
    this))


(defn factory
  [{:keys [port] :as config}] ⑯
  (hcomp/make-component ::server (start! port) stop!)) ⑰
```

① Namespace definition. As you can see we orginized the `web` component under `getting-started.proxy.components` namespace. It is a good practice to store all of your components under the same parent namespace.

② We're going to use the awesome http://aleph.io(Aleph) library to build our webserver. HellHound depends on this library by default (If we use the `codamic/hellhound` or `codamic/hellhound.http` artifact).

③

`manifold.stream` library provides a very robust stream abstraction. Component's IO build around this abstraction and we need to use this library to work with component's IO. HellHound depends on the `manifold` library, so there's no need to add it explicitly to your dependencies.

④ `manifold.deferred` library provides an abstraction around promises and futures, and aleph webserver uses them as async responses.

⑤ The return value of this function is in fact the ring handler which we want to use to handle the incoming requests. The two paramter of `handler` function are the `input` and `output` of the webserver component.

⑥ Before we return the actual ring handler, we need to setup a consumer for incoming data from the input stream. The basic idea is to treat any incoming data from the `input` stream of webserver component as a response and return the response to the user. Alph support async responses using deferreds, So by returning a deferred from **(8)** alph send back the response as soon as the deferred realised. In our consumer we just extract the same deferred value from the incoming map (any incoming value is a hashmap and we placed the deferred into it in **(9)**) and resolve it the response map that again extracted from the same hashmap.

⑦ The actual ring handler that receives the HTTP request hashmap (`req`).

⑧ Created a deferred value to use as the response and pass it through the pipeline.

⑨ Simply create a hashmap containing `:request` and `:response-deferred` with the corresponding values and put the hashmap into the webserver output stream (send it down to the pipeline). Then, return the created deferred so alph can return the reponse to user as soon as the response deferred realise.

⑩ Receives a port number and return an anonymous function to be used as a start function for webserver component. It passes the given port number to the `start-server` function of aleph.

⑪ `hellhound.component/io` function is a helper function which returns a vector in form of `[input output]` of the IO streams of the given component.

⑫ Starts the aleph webserver. As you can see we passed the input and output of the component to `handler` fn. We assigned the returned value of `start-server` to a key in component map so we can close it later.

⑬ `stop-fn` of the webserver component.

⑭ Stop the aleph server by calling `.close` on the server object which we assigned to `:server` in the `start-fn`.

⑮ Remove the `:server` key to get back to the initial state.

⑯ Desctruct the port number from the given `config` map.

⑰ Create and return a component map for with the name of `::server`.

So far we created a webserver component which its output is a stream of hashmaps that each of them has a `:request` key containing the request map and conceptually it will send any incoming response from the input stream to user.

Now we need another component to fetch the content of each requested URL from the remote host. Let's jump in:

Crawler component.

```
(ns getting-started.proxy.components.crawler
  (:require
   [aleph.http :as http]   ①
   [manifold.deferred :as d]
   [manifold.stream :as stream]
   [hellhound.component :as component]))



(defn response     ②
  [res]
  {:status (:status res)
   :headers (:headers res)
   :body (:body res)})

(defn fetch-url
  [url]
  (println "PROXYING '" url "'...")
  (d/chain (http/get url)    ③
           response))

(defn proxy   ④
  [output host]
  (fn [event]
    (let [request (:request event)
          url     (str host (:uri request))]
      (stream/put! output                          ⑤
                   (assoc event
                          :response
                          (fetch-url url))))))    ⑥

(defn start
  [host]
  (fn [this context]
    (let [[input output] (component/io this)]     ⑦
     (stream/consume (proxy output host) input)   ⑧
     this)))

(defn stop [this] this)   ⑨

(defn factory
  [host]
  (component/make-component ::job (start host) stop))  ⑩
```

① Yupe, aleph again. Aleph library provides a HTTP client as well.

② A simple an stupid function to extract data from the response and generate a ring response again. This function is useless in this case but You might want to change the response a bit. In that case this function would be the best place for your logic.

③ Fetch the content of the given url and uses the `manifold.deferred/chain` function to pass the content to `response` function as soon as it arrived. `manifold.deferred/chain` returns a `deferred` value.

④ Returns an anonymous function with gets a parameter that would be any incoming value from the input stream. Then, we we extract the request from the incoming value and fetch the content for that request.

⑤ Fetch the content or `url` by using `fetch-url` function and creates a new hashmap from the received input data with an additional key called `:response` that has a deferred value representing the content and push it to the output of crawler component.

⑥ Retrieve the content of the given `url`. The return value of this function is a deferred value.

⑦ `hellhound.component/io` is a helper function that returns input and output of the given component.

⑧ Consumes any incoming value from the input and uses the function returned by the `proxy` function to process each value.

⑨ A dummy stop funtion which does nothing. Because we don't have any termination or cleanup logic.

⑩ Creates a new component with name `:getting-started.proxy.components.crawler/job`.

Alright, Now we have two components that we can create a system from them and build our proxy.

2.7. System

HellHound's systems are the most important part of our application. We express our thoughts on how the application should behave using systems.

Let's create the first version of our system and then talk about a little bit of details.

Crawler component.

```
(ns getting-started.proxy.system
  (:require
   [getting-started.proxy.components.web :as web] ①
   [getting-started.proxy.components.crawler :as crawler])) ②


;; System factory
(defn factory ③
  [port host]
  {:components                          ④
        [(web/factory {:port port})     ⑤
         (crawler/factory host)]        ⑥

   :workflow                                ⑦
        [[::web/server ::crawler/job]       ⑧
         [::crawler/job ::web/server]]})    ⑨
```

① Our webserver component.

② Our crawler component.

③ A system factory function which gets the configuration of the system and returns a system map based on those values.

④ Each system must have a `:components` key with a collection of components as the value.

⑤ We created an instance of webserver component by calling the factory function of web namespace and passing the webserver configuration to it. We can refer to this component later using its name which we defined in the web ns.

⑥ Same as webserver component we created an instance of the crawler.

⑦ Every system must have a `:workflow` key as well. This key describes how HellHound should create the data pipeline by pipeing IO of components together. Its value should be a collection of collections which each element represent a connection.

⑧ In this connection we want to plug the output stream of `::web/server` (which is the name of our webserver component) to input stream of `::crawler/job` (which is the name of our crawler component). So any value that goes to `::web/server` output would be available to `::crawler/job` through its input stream.

⑨ In this connection we connect `::crawler/job` back to `::web/server`. So values from crawler's output will goes to webserver input stream.

As you can see, system definition is fairly straight forward. HellHound starts all the components by calling their start function first and the setup the data pipeline based on the `:workflow` description.

In our example by sending a request to our webserver, the `::web/server` component will process the request and send to to the its output and the request would endup in the input of `::crawler/job`. Then crawler fetches the content of any incoming request and put it to its output that according to the `:workflow`, its output connected to the input of `::web/server`. Webserver component will send back in value coming from the input stream to user as response. Pretty simple, isn't it ?

Systems decouple the logic of the program in to smaller pieces (components). It's easier to reason about each piece and it's much easier to replace a piece with another piece. Due to input/output abstraction systems are highly composable and components are heavily reusable.

Now it's time to use our system and run our program. Here is our `core` namespace and main function:

Core namespace.

```
(ns getting-started.proxy.core
  (:require [getting-started.proxy.system :as system]
            [hellhound.system :as hellhound]
            [aleph.http :as http])
  (:gen-class))


(defn -main
  [& args]
  (let [[url _port path] args
        port             (Integer. (or _port "3000"))]

    (if (nil? url)
      (println "URL is missing. Please pass the 3rd party URL as the first arg")
      (let [proxy-system (system/factory port url)]       ①
        (println (str "Starting server on " port "..."))
        (hellhound/set-system! proxy-system)              ②
        (hellhound/start!)))))                             ③
```

① Create an instance of our system by passing the configuration to the factory function.

② Set the default system of our program. Each program can have only one default system at any given time.

③ Fire up the default system (start the program).

Now it's time to run our program using lein. Issue the following command in the project root:

```
lein run -m getting-started.proxy.core/-main http://hellhound.io 3000
```

Or if you're using your own project:

```
lein run http://hellhound.io 3000
```

Open up your browser and navigate to http://localhost:3000/(http://localhost:3000/). You should be able to see HellHound's website loading using your proxy.

2.8. Version 2.0

Let's improve our code so we can serve the index of the website locally and through a file (Actually I personally use this software on my daily development). But first, let's think about what we want to do. Generally in order to do such thing, we should look into the `:uri` value inside the request map of a ring handler. If it is `/` we need to load the content from a file instead of sending a request to the remote host. We need to treat other URIs the same as before.

We can use system's workflow to achieve our goal. But we need more components. Let's create a component which loads the index content from a file.

Index loader component

```
(ns getting-started.proxy.components.index-loader
  (:require
   [manifold.stream :as stream]
   [hellhound.component :as component]))


(defn response
  [body]
  {:body body :status 200 :headers []})


(defn load-index   ①
  [output]
  (fn [event]      ②
    (let [path (System/getProperty "user.dir")  ③
          file (str path "/index.html")]        ④
      (println (str "Loading: " file))

      (stream/put!
       output
       (assoc event :index-content (slurp file)))))) ⑤

(defn start!
  [this context]
  (let [[input output] (component/io this)]
    (stream/consume (load-index output) input)   ⑥
    this))

(defn stop [this] this)

(defn factory
  []
  (component/make-component ::job start! stop))  ⑦
```

① Returns a function that process any incoming value from the input and but the result to output stream

② `event` will hold each of the incoming values.

③ Get the home directory of the user.

④ Set the file path to `$HOME/index.html`

⑤ Reads the content of the `file` and assign it to a new key in the map. Send the map to the output.

⑥ Consumes any incoming value and process them using the anonymous function returned by `load-index` fn.

⑦ Creates a component and call it `::job`.

Now just for fun let's create another component which reads from the input stream and construct a ring response from any incoming value and put the generate ring response to the output stream. Totally simple and easy to build.

Response generator component

```
(ns getting-started.proxy.components.response
  (:require [hellhound.component :as component]
            [manifold.stream :as stream]))



(defn make-response
  [output]
  (fn [{:keys [index-content] :as event}]
    (stream/put! output
                 (assoc event :response {:body index-content
                                         :headers []
                                         :status 200}))))

(defn ->response
  [this context]
  (let [[input output] (component/io this)]
    (stream/consume (make-response output) input)
    this))

(defn ->response-factory
  []
  (component/make-component ::->response ->response #(identity %))) ①
```

① Let's call this component `::→response`.

As you see, the response generator component is very very simple.

Now It is time to improve our system and workflow.

Response generator component

```
(ns getting-started.proxy.system
  (:require
   [getting-started.proxy.components.web :as web]
   [getting-started.proxy.components.crawler :as crawler]
   [getting-started.proxy.components.index-loader :as loader]
   [getting-started.proxy.components.response :as response]))

(defn uri     ①
  [event]
  (:uri (:request event)))

(defn factory
  [port host]
  {:components [(web/factory {:port port})
                (crawler/factory host)
                (loader/factory)
                (response/->response-factory)]

   :workflow [[::web/server #(not (= "/" (uri %))) ::crawler/job]   ②
              [::web/server #(= "/" (uri %))       ::loader/job]    ③
              [::crawler/job ::web/server]                          ④
              [::loader/job  ::response/->response]                 ⑤
              [::response/->response ::web/server]]})               ⑥
```

① A helper function that extract the uri from the ring request map.

② In this connection we want all the values which their `:uri` value is not `/` to go to `::crawler/job` component from `::web/server`. The second value in the connection collection (if total number of values in the connection collection is more than two) is a predicate function. If that predicate returns true only then values will pass to downstream component.

③ By the predicate of this connection, only requests which has `/` as the `:uri` value passes down to the downstream component that is `::loader/job`.

④ Same as before, we want the output of `::crawler/job` to be connected to input of `::webserver`.

⑤ Connects the output of `::loader/job` to input of `::response/→response`. This step might be unnecessary we could just do it in the same component but just to have more complicated workflow, we separated the components.

⑥ And finally the output of `::response/→response` component (which will be a stream of ring responses) should goes to `::web/server`.

In this example we branched off our workflow from `::web/server` based on couple of predicates functions. Each branch will do its job and will have its own workflow. Finally we merge both pipes back in `::web/server` again. So any request flowing from webserver that is not a request to get the root path will goes to `::crawler/job` and the content would be fetched from the actual host and the request for root path will goes to `::loader/job` and loads from a file on filesystem. Pretty neat, right?

I hope this tutorial gave you an idea about HellHound systems. HellHound is definitely more than this, but now you have enough knowledge to get started with **HellHound**.

Chapter 3. Systems

At heart, **HellHound** created around the idea of systems. The basic idea is to describe the execution model and dataflow of a program using data and let **HellHound** handles the rest. Systems are the first concept of **HellHound** which you need to learn about.

3.1. Overview

Let’s take a quick conceptual look at **HellHound Systems**. As you already know **HellHound System** is available under `codamic/hellhound.core` and `codamic/hellhound` artifacts.

3.1.1. The Unix Philosophy

Contemporary software engineering still has a lot to learn from the 1970s. As we’re in such a fast-moving field, we often have a tendency of dismissing older ideas as irrelevant—and consequently, we end up having to learn the same lessons over and over again, the hard way. Although computers have become faster, data has grown bigger, and requirements have become more complex, many old ideas are actually still highly relevant today.

In this section, I’d like to highlight one particular set of old ideas that I think deserves more attention today, the Unix philosophy.

Unix pipeline

The Unix philosophy is a set of principles that emerged gradually during the design and implementation of Unix systems during the late 1960s and 1970s. There are various interpretations of the Unix philosophy, but in the 1978 description by Doug McIlroy, Elliot Pinson, and Berk Tague, two points particularly stand out:

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new “features.”
- Expect the output of every program to become the input to another, as yet unknown, program.

These principles are the foundation for chaining together programs into pipelines that can accomplish complex processing tasks. The key idea here is that a program does not know or care where its input is coming from, or where its output is going: it may be a file, or another program that’s part of the operating system, or another program written by someone else entirely.

Pipes and Composability

The tools that come with the operating system are generic, but they are designed such that they can be composed together into larger programs that can perform application-specific tasks.

The benefits that the designers of Unix derived from this design approach sound quite like the ideas of the Agile and DevOps movements that appeared decades later: scripting and automation, rapid prototyping, incremental iteration, being friendly to experimentation, and breaking down large projects into manageable chunks.

When you join two commands by using the pipe character in your shell, the shell starts both programs at the same time, and attaches the output of the first process to the input of the second process. This attachment mechanism uses the pipe syscall provided by the operating system.

Note that this wiring is not done by the programs themselves; it's done by the shell—this allows the programs to be loosely coupled, and not worry about where their input is coming from or where their output is going.

The pipe had been invented in 1964 by Doug McIlroy, who described it like this in an internal Bell Labs memo, "We should have some ways of coupling programs like [a] garden hose—screw in another segment when it becomes necessary to massage data in another way."

The Unix team also realized early that the interprocess communication mechanism (pipes) can look very similar to the I/O mechanism for reading and writing files. We now call this input redirection (using the contents of a file as input to a process) and output redirection.

The reason that Unix programs can be composed so flexibly is that they all conform to the same interface. Most programs have one stream for input data (`stdin`) and two output streams (`stdout` for regular output data, and stderr for errors and diagnostic messages to the user).

Programs can also do other things besides reading stdin and writing stdout, such as reading and writing files, communicating over the network, or drawing a graphical user interface. However, the stdin/stdout communication is considered to be the main means for data to flow from one Unix tool to another.

The great thing about the stdin/stdout interface is that anyone can implement it easily, in any programming language. You can develop your own tool that conforms to this interface, and it will play nicely with all the standard tools that ship as part of the operating system.

Composability Requires a Uniform Interface

We said that Unix tools are composable because they all implement the same interface of `stdin`, `stdout`, and `stderr`, and each of these is a file descriptor; that is, a stream of bytes that you can read or write like a file. This interface is simple enough that anyone can easily implement it, but it is also powerful enough that you can use it for anything.

Because all Unix tools implement the same interface, we call it a uniform interface. That's why you can pipe the output of `gunzip` to `wc` without a second thought, even though those two tools appear to have nothing in common. It's like lego bricks, which all implement the same pattern of knobbly bits and grooves, allowing you to stack any lego brick on any other, regardless of their shape, size, or color.

A few tools (e.g., `gzip`) operate purely on byte streams and don't care about the structure of the data. But most tools need to parse their input in order to do anything useful with it. For this, most Unix tools use ASCII, with each record on one line, and fields separated by tabs or spaces, or maybe commas.

HellHound & Unix Philosophy

We've seen that Unix has developed good design principles for software development. HellHound's systems mostly follow the same design principle. There are several differences though. A Unix pipe is designed to have a single sender process and a single recipient. You can't use pipes to send output to several processes, or to collect input from several processes. But HellHound Systems are not like this. A Component (process) can pipe its output to more than one component and accept its input from more than one component.

Instead of reading/writing data in text, components communicate via stream abstractions delivering data structures like hashmaps.

Unix processes are generally assumed to be fairly short-running. For example, if a process in the middle of a pipeline crashes, there is no way for it to resume processing from its input pipe—the entire pipeline fails and must be re-run from scratch. That's no problem if the commands run only for a few seconds, but if an application is expected to run continuously for years, you need better fault tolerance. Systems in the other hand design for long-running tasks. A component assumes to run for a long time and constantly consume from its input and product to its output stream. But still if a component fails it's possible to restart the component and re-pipe it, so it can continue where it left off. But right now there is no support for restarting a component. We plan to add the support for version 1.1.0 (hopefully).

3.1.2. Execution model

Systems composed by [Components]. A system knows how to start and stop components. It is also responsible for managing dependencies between theme. Components are the smallest parts of a system whic are reusable.

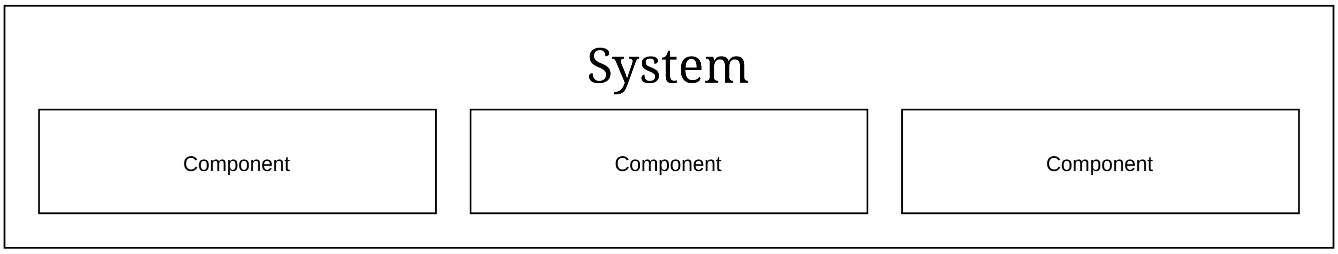

Figure 1. A very basic schema of a System

A **HellHound** system is basically a `map` describing different aspects of a program. Each program might have several systems. For example a development system and a production system. But there would be just a system running at a given time.

Systems should follow a certain spec (`:hellhound.system/system`). For example any system should have a`:components` key with a vector as its value. All the `components`of the system should be defined under this key. Each component is a `map` describing the behaviours of that component (For more info on components read [here](./Components.md)).

In order to use **HellHound** systems. First you need to define your system by defining a map with at least one key. Yupe, you guessed it correctly, the `:components` key. We're going to discuss the system's keys in bit. So for now just let's talk about how to start and stop a system.

After defining the system. The next step would be to set the defined system as the default system of your program by using `hellhound.system/set-system!` function. This function accepts just one argument which is the system map. It basically analayze the given map and set it as the default system of the program. From now on, you can call `hellhound.system/start` and `hellhound.system/stop` function.

Basic execution flows of a system are `start` and `stop`. Both of them creates a graph from the system components and their dependencies. Then they `start`/`stop` the system by walking through that graph and calling the `start-fn` and `stop-fn` function of each component.

After starting a system, you can access the running system via `hellhound.system/system` function. It would returns a map describing the current running system including all the running components.

There are two ways to manage dependencies in your components code. The first one via a the system map itself, by treating the system map as a repository of dependencies. The second way is via dependency injection model which is not supported at this version (`v1.0.0`). Anyway, any component can pull what ever it needs from the system map by calling `hellhound.system/get-component` function and passing the component name to it. This function will simply look for the component in system map and returns the running component map. There is a huge problem with this apparoach. By using this apparoach components knows too much about the environment around them, so it reduce their portablity.

In near future we’re going to add the support for dependency injection to **HellHound** systems.

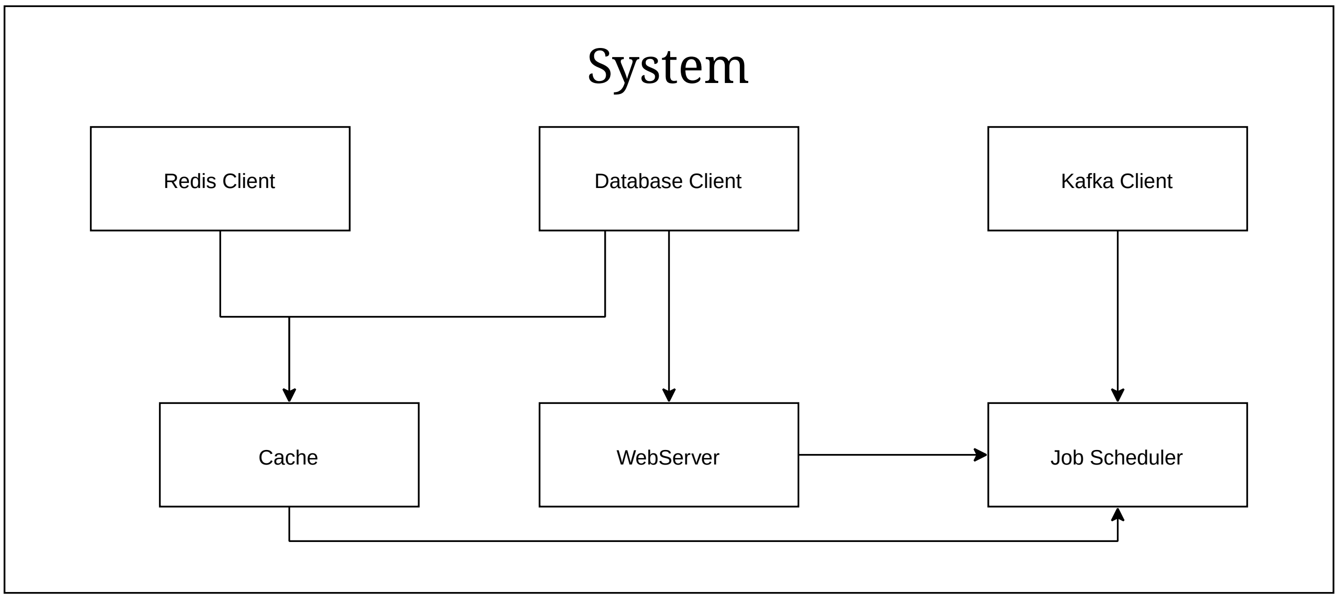

Figure 2. A schema of components of a system and how they might depends on each other

3.1.3. Workflow

System’s workflow describes the dataflow of the system. It describes where data comes in and where information goes out of the system. By "where" I mean which component of the system. As mentioned before each **HellHound** [Component](./Components.md) has an **input stream** and an **output stream** assigned to them. You can think of each component as a pipe, the whole system as a pipeline. Data flows to this pipeline based on your piping (description) from entry points and exits the pipeline from exit points. Pipeline might have more than one entry or exit points or none at all.

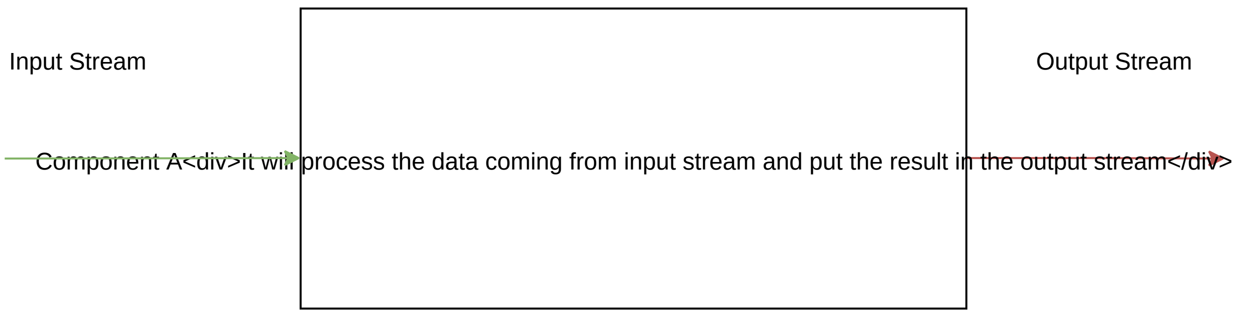
Input Stream
Output Stream
Component A<div>It will process the data coming from input stream and put the result in the output stream</div>

Figure 3. Components process the input stream and produce the output put stream

Using the system workflow you can design an open or a close dataflow for your system. A Close system is a type of system which all of its components have their **input** and **output** connected. In the other hand, an open system is a type of system which **NOT** all of the **inputs** and **outputs** of components connected to each other.

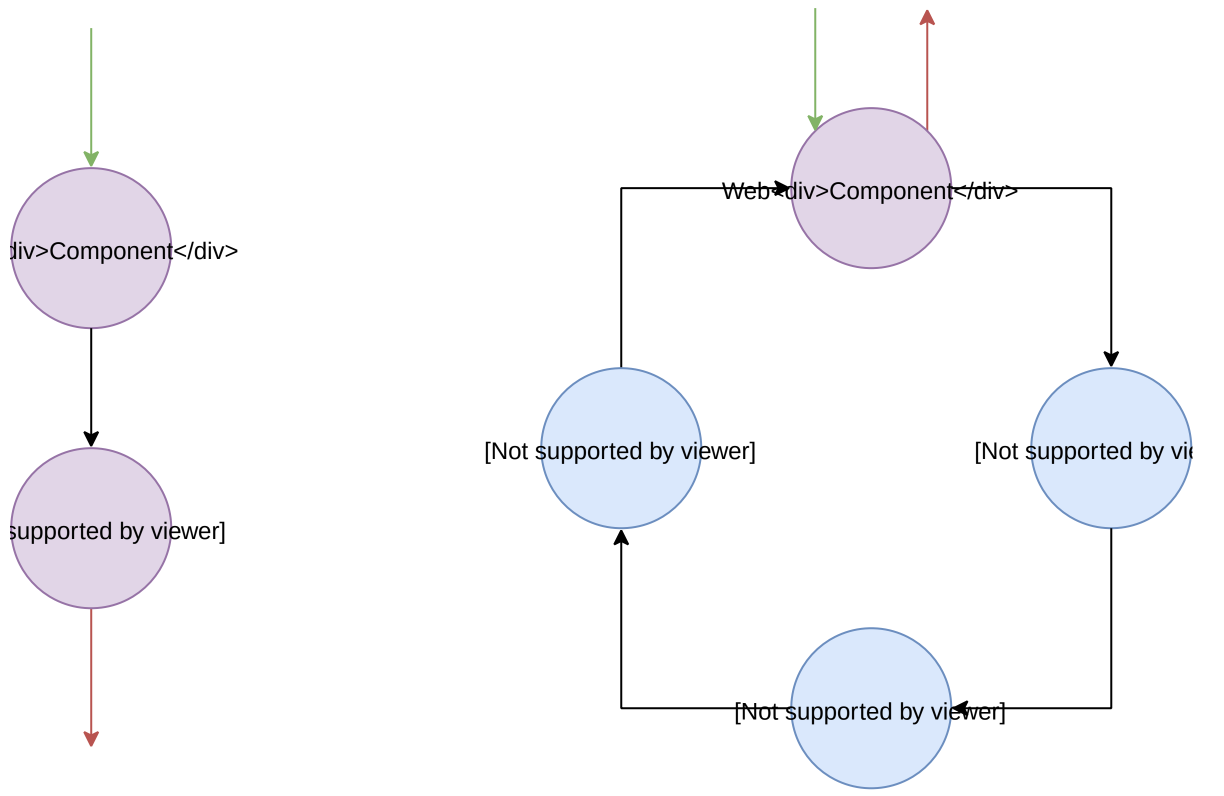

Figure 4. Open system on the left and Close system on the right

❗ *Don't confuse Input and Output of each component which components dependencies.*

Components of a system should consum from their `INPUT` and produce their `OUTPUT` in non-blocking fashion in order to avoid blocking in a system.

Check out the [workflow](./Workflow.md) section for more information.

3.2. Components

A **Component** is a data structure to managing the lifecycle, dependencies and data flow of a program. A Component is the smallest unit of execution which is reusable and easy to reason about.

A component is similar in spirit to the definition of an object in Object-Oriented Programming. This does not alter the primacy of pure functions and immutable data structures in Clojure as a language. Most functions are just functions, and most data are just data. Components are intended to help manage stateful resources within a functional paradigm.

3.2.1. Advantages of the Component Model

Large applications often consist of many stateful processes which must be started and stopped in a particular order. The component model makes those relationships explicit and declarative, instead of implicit in imperative code.

Components provide some basic guidance for structuring a **HellHound** application, with boundaries between different parts of a system. Components offer some encapsulation, in the sense of grouping together related entities. Each component receives references only to the things it needs, avoiding the unnecessary shared state. Instead of reaching through multiple levels of nested maps, a component can have everything it needs at most one map lookup away.

Instead of having mutable state (atoms, refs, etc.) scattered throughout different namespaces, all the stateful parts of an application can be gathered together. In some cases, using components may eliminate the need for mutable references altogether, for example, to store the "current" connection to a resource such as a database. At the same time, having all state reachable via a single [**system**](./README.md#overview) map makes it easy to reach in and inspect any part of the application from the REPL.

The component dependency model makes it easy to swap in **stub** or **mock** implementations of a component for testing purposes, without relying on time-dependent constructs, such as with-redefs or binding, which are often subject to race conditions in multi-threaded code.

Having a coherent way to set up and tear down all the state associated with an application enables rapid development cycles without restarting the JVM. It can also make unit tests faster and more independent, since the cost of creating and starting a system is low enough that every test can create a new instance of the system.

3.2.2. Disadvantages of the Component Model

For small applications, declaring the dependency relationships among components may actually be more work than manually starting all the components in the correct order or even not using component model at all. Everything comes at a price.

The [**system**](./README.md#overview) map produced by **HellHound** is a complex map and it is typically too large to inspect visually. But there are enough helper functions in `hellhound.system`` namespace to help you with it.

You must explicitly specify all the dependency relationships among components: the code cannot discover these relationships automatically.

Finally, **HellHound** system forbids cyclic dependencies among components. I believe that cyclic dependencies usually indicate architectural flaws and can be eliminated by restructuring the application. In the rare case where a cyclic dependency cannot be avoided, you can use mutable references to manage it, but this is outside the scope of components.

3.2.3. Usage

Components are the main parts of HellHound systems. Basically, each component is an implementation of `IComponent` protocol. The protocol which defines a component functionality. By default HellHound implements `IComponent` protocols for hashmaps only. So we can define components in form of maps.

In order to define a component, a map should contain the following keys (All the keys should be namespaced keyword under `hellhound.component`):

- **`name`**: The name of the component. It should be a namespaced keyword. This key is **mandatory**.
- **`depends-on`**: This key specifies all the components which are the dependencies of the current component. A collection of components name. This key is optional.
- **`start-fn`**: A function which takes which takes two arguments. The component map and a context map. Returns the component with the necessary keys attached to it. This function is responsible for **starting** the component. Practically your component code goes into this function. This key is **mandatory**.
- **`stop-fn`**: A function which takes the component map as the only argument and return the component with the necessary keys attached to it. This function is responsible for **stoping** the component. This key is **mandatory**.
- **`input-stream-fn`**: A function which returns a `manifold` as the input of the component. You rarely need to use this key for a component. This key optional.
- **`output-stream-fn`**: A function which returns a `manifold` as the output of the component. You rarely need to use this key for a component. This key optional.

So as an example:

A Stupid component which does nothing

```
(def sample-component
  {:hellhound.component/name :sample.core/component-A ①
   :hellhound.component/start-fn (fn [component] component) ②
   :hellhound.component/stop-fn (fn [component] component)  ③
   :hellhound.component/depends-on [:sample.core/component-B]}) ④
```

① Description of the component name which is `:sample.code/component-A`

② The function which is responsible for starting the component. Which does nothing in this case.

③ The function which is responsible for stapping the component. Which does nothing in this case.

④ Description of the dependencies of `:sample.code/component-A`. In this case this component depends on another component called `:sample.core/component-B`.

As you can see creating a component is fairly simple and nothing special.

3.2.4. Context Map

As you already know, start-fn of a component takes two arguments. The first one it the component map itself and second one is a hashmap called **context map**.

The purpose of a context map is to provides co-effects for the component, Such as dependency components. For example **HellHound** inject a key called `:dependencies` to the context map which is a vector containing those components that this component is depends on. The order of this vector would be exactly the same as :depends-on value in component description map. So you can get all you need from the context map. You can think of it as some sort of dependency injection for you component.

But context map contains more data that you might need them in action. Here is the list of keys of a context map:

- **`:dependencies`**: A vector of running components which the current component is depends on. All the components in this vector already started and the order is the same as the vector provided to :depends-on in component map.
- **`:dependencies-map`**: A hashmap containing all the components of the current component dependencies. keys of this map are the components names and the values are the running components. The purpose of this map is to access dependency components by name.

3.3. Workflow

System's workflow a vector describing the dataflow of the system. Each component has an input and an output stream. Each stream is a`manifold.stream`. HellHound connects io of each component to another component based on the desciption given by the `:workflow` of the system.

System's workflow is a vector of vectors. Each vector contains two mandatory element which are:

- The name of the output component
- The name of the input component

For example, consider the following system:

Snippet 1 - A HellHound system

```
(ns example.system
  ;; We want to use defcomponenet to define the components instead of defining the
maps
  (:require [hellhound.system :as hh]
            [hellhound.system :as hcomp]
            [manifold.stream :as s]))

(defn dummy-start ①
  [component ctx]
  (let [input  (hcomp/input  component)
        output (hcomp/output component)])
  ;; Simply direct incoming data to output
  (s/connect input output)
  component)

(defn dummy-stop [component] component)

(def example-system
     {:components [(hcomp/make-component :component-1 dummy-start dummy-stop)   ②
                   (hcomp/make-component :component-2 dummy-start dummy-stop)   ③
                   (hcomp/make-component :component-3 dummy-start dummy-stop)]  ④

      :workflow [[:component-1 :component-2]    ⑤
                 [:component-2 :component-3]]}) ⑥


(hh/set-system! example-system) ⑦
(hh/start!) ⑧
```

① A dummy start function which connects input to output directly

② Definition of `component-1`

③ Definition of `component-2`

④ Definition of `component-2`

⑤ Description to connect output of `component-1` to input `component-2`

⑥ Description to connect output of `component-2` to input `component-3`

⑦ Sets `example-system` as the current system

⑧ Starts the current system.

In the above snippet we defined three different components. For the sake of simplicity I skipped the start and stop function defination of each component.

After starting the system defined in the above snippet, **HellHound** wires the input/output of each component as we described in the system. So the running system workflow would be like:

[Workflow 1]

```
Data --> Component 1 ---> Component 2 ---> Component 3 ---> ...
```

As you can see in the above flow, the output of `component-3` goes to no where. It's not necessary for a component to has both input and output connected to somewhere. Some components are responsible for side effects. For example writing to DB, Send data to client or dispatch an event to a kafka topic. These kind of components may not have an output stream. An other example would be those components which provide data to the pipeline by reading from a DB or from an external stream like a kafka stream of client side event stream and so on. These components do not need an input stream to operate.

So in the `example-system` we can test the workflow simply like this:

Snippet 2 - An example of a working workflow

```
(let [input1 (hcomp/input (hh/get-component :component-1))
      output3 (hcomp/output (hh/get-component :component-3))]
  (s/put! input1 42) ①
  (println "output: " @(s/take! output3))) ;; ==> output: 42  ②
```

① Put number `42` into the pipeline enterance point

② Take the output from the output point of the pipeline

Since we we connected our pipeline as described in **workflow 1**, The value which we put into input stream of `component-1`, retreived from the output stream of `component-3`.

Don't forget that our components simply connect their input stream to their output stream

Now let's change our workflow to create a tree like pipeline. Now let's change the `:workflow` of `example-system` as follow:

```
:workflow [[:component-1 :component-2]
           [:component-1 :component-3]]
```

Now with this new workflow, our dataflow would be like:

[Workflow 2]

```
                          |---> Component 2 ---> ...
Data --> Component 1 --> |
                          |---> Component 3 ---> ...
```

So both of `component-2` and `component-3` would get their input from `component-1`. Any input comes to `component-1` would be available separately in the output of `component-2` and `component-3` and taking that value from the output of `component-2` would not affect the output of `component-3`.

3.3.1. Conditional Workflow

Sometimes, we need to dispatch values to a component conditionally. For example imagin a system that is responsible for separeting odd numbers from even numbers in a stream of numbers. Checkout the following `workflow` definition:

```
:workflow [[:component-1 odd?  :component-2]
           [:component-1 even? :component-3]]
```

As you can see in the workflow definition above, it’s possible to describe a condition for dispatching values from a component to another one. In the previous example all the values from `:component-1` flow to `:component-2` only if the provided condition returns true for each incoming value which in this case only `odd` values are going to deliver to `component-2`. The same applies to the second pipe but only `even` values flow to `component-3`.

The predicate function can be any function which receives an argument (The value from upstream component) and returns a boolean value indicating whether the value should flow to the downstream component or not.

Base on what we discussed up until now we can test our workflow like this:

Snippet 3 - Testing conditional workflow

```
;; I skipped the ns declaration. It would be exactly like Snippet 1 in this section
(let [input1 (hcomp/input (hh/get-component :component-1))
      output2 (hcomp/output2 (hh/get-component :component-2))
      output3 (hcomp/output2 (hh/get-component :component-3))]

  (s/consume #(println "Odd: " %) output2) ①
  (s/consume #(println "Even: " %) output3) ②

  (-> [1 2 3 4 5]
      (s/->source) ③
      (s/connect input1))) ④
```

① Adds a consumer function for the output stream of `:component-2`

② Adds a consumer function for the output stream of `:component-3`

③ Converts the `[1 2 3 4 5]` vector to a stream source

④ Connects the source stream resulted in step <3> to input of `:component-1`

The output of the above snippet would be like:

```
Odd: 1
Even: 2
Odd: 3
Even: 4
Odd: 5
```

❗ *Predicate functions should be pure*

Predicate functions in each pipe should be pure and free of side effects. These functions should be as fast as possible because **HellHound** calls them rapidly for each value in the pipe.

<PLACEHOLDER TEXT> Explaination about predicate best practices and `hellhound.messaging` ns

3.4. Examples

Now, let’s take a look at some examples about how systems works and let’s take what we learned up until now into an action. All the examples are available at the hellhound_examples repository on github.

3.4.1. Very Simple System

First of all let’s create a very simple component with just one dependency to have a minimum working environment.

In order to execute the following example from the hellhound_examples repository. Just issue the following command in the repository root directory.

```
$ lein run -m systems.simple-system1/main
```

Ok let’s jump into the code.

```
(ns systems.simple-system1
  (:require [hellhound.system :as system]
            [hellhound.component :as hcomp]))

(defn start-fn1 ①
  [component context]
  (println "Starting Component 1...")
  (assoc component :something-in "Hello World")) ②

(defn stop-fn1 ③
  [component]
  (println "Stopping component 2...")
  component)

(defn start-fn2 ④
  [component context]
  (let [component1 (first (:dependencies context)) ⑤
        component1-with-name (:simple-system/component-1 (:dependencies-map context))]
⑥
    (println "Starting Component 2...")
    (println (:something-in component1)) ⑦
    (println (:something-in component1-with-name))
    component))

(defn stop-fn2
  [component]
  (println "Stopping component 2...")
  component)

(def component-1 {:hellhound.component/name :simple-system/component-1 ⑧
                  :hellhound.component/start-fn start-fn1
                  :hellhound.component/stop-fn  stop-fn1})

(def component-2 (hcomp/make-component :simple-system/component-2
                             start-fn2
                             stop-fn2
                             [:simple-system/component-1])) ⑨

(def simple-system ⑩
  {:components [component-2 component-1]})

(defn main
  []
  (system/set-system! simple-system) ⑪
  (system/start!)) ⑫
```

① The start function of the component 1 which will assign to the component later in the component map. The first argument is the componet map itself and the argument is the context map which contains extra info about the context which this component is running on. As you already know start function of a component should return a component map.

② Assinged a simple value to a key in component map. So the running component would have this value attached to it and other components can use the value by getting it from the component map.

③ Stop function of the `component1`. It should returns a component map.

④ Start function of `component-2`.

⑤ Gets the first dependency component. In the order which defined in the component map `:depends-on`

⑥ Gets the same component by it's name instead.

⑦ Gets a value from another component.

⑧ A component map which defines the `:simple-system/component-1` component Intentionally we defined this component by defining a map directly. But you can use `make-component` function as a shortcut.

⑨ We used `make-component` to define a component called `:simple-system/component-2`. It basically returns a map like we had in component-1 with the given details.

⑩ A very simple `system` defination which does not have any workflow. Please not that the order of components in `:components` vector is **NOT** important at all.

⑪ Setting the `simple-system` as the default system of the application.

⑫ Start the default system.

> The last argument is the dependency vector of the component that exactly is going to be the `:depends-on` key in the component map.

When we execute the above code the output would be something like:

The output of the above namespace

```
[17-10-29 12:27:37] [DEBUG] <hellhound.component:115> - Starting component ':simple-
system/component-1'... ①
Starting Component 1... ②
[17-10-29 12:27:37] [DEBUG] <hellhound.component:115> - Starting component ':simple-
system/component-2'...
Starting Component 2... ③
Hello World ④
Hello World ⑤
[17-10-29 12:27:37] [DEBUG] <hellhound.component:120> - Component ':simple-
system/component-1' already started. Skipping... ⑥
[17-10-29 12:27:37] [INFO] <hellhound.system.core:142> - System started successfully.
⑦
[17-10-29 12:27:37] [DEBUG] <hellhound.system.workflow:104> - Setting up workflow...
[17-10-29 12:27:37] [INFO] <hellhound.system.workflow:107> - Workflow setup done.
```

① A log entry with `DEBUG` level states that HellHound is going to start `:simple-system/component-1`

② The output of the `println` in line of [6] of the above code.

③ The output of the `println` in line of [18] of the above code.

④ The output of the `println` function in the start function of th `:simple-system/component-2`

⑤ Same as number 4.

⑥ A log entry which states that `:simple-system/component-1` is already running and HellHound is going to skipp it.

⑦ System started at this point completely.

3.4.2. Simple workflow

In this example we're going to create a really simple system with a simple linear workflow. The workflow is really simple, data will flow from `component-1` to `component-2` and then `component-3`. Each component transoform the value and put it to the output.

In order to execute the following example from the hellhound_examples repository. Just issue the following command in the repository root directory.

```
$ lein run -m systems.simple-system2/main
```

Ok let's jump into the code.

```
(ns systems.simple-system2
  (:require
   [manifold.stream :as s]
   [hellhound.system :as system]
   [hellhound.component :as hcomp]))

(defn start-fn1 ①
  [component context]
  (let [input  (hcomp/input component)
        output (hcomp/output component)]

    (s/consume (fn [x]
                 (println "Message Component 1: " x)
                 (s/put! output (inc x)))
               input))
  component)

;; Stop function of all the components. It should returns a component
;; map.
(defn stop-fn
  [component]
  component)

;; Start function of component-2.
(defn start-fn2
  [component context]
  (let [input (hcomp/input component) ②
        output (hcomp/output component)] ③
    (s/connect-via input
                   (fn [x]
                     (println "Message Component 2: " x)
                     (s/put! output (inc x)))
                   output) ④
    component))

;; Start function of component-2.
(defn start-fn3
  [component context]
  (let [input (hcomp/input component)]
    (s/consume #(println "Message Component 3: " %) input) ⑤
    component))

(def component-1 (hcomp/make-component :simple-system/component-1 start-fn1 stop-fn))
⑥
(def component-2 (hcomp/make-component :simple-system/component-2 start-fn2 stop-fn))
(def component-3 (hcomp/make-component :simple-system/component-3 start-fn3 stop-fn))

(def simple-system
  {:components [component-2 component-1 component-3]
   :workflow [[:simple-system/component-1 :simple-system/component-2] ⑦
              [:simple-system/component-2 :simple-system/component-3]]})

(defn main
  []
  (system/set-system! simple-system)
  (system/start!)


  (let [component1 (system/get-component :simple-system/component-1) ⑧
        input      (hcomp/input component1)]

    (-> [1 2 3 4 5 6]
        (s/->source) ⑨
        (s/connect input))) ⑩
  (println "Done."))
```

① The start function of the component 1 which will assign to the component later in the component map. The first argument is the componet map itself and the argument is the context map which contains extra info about the context which this component is running on. As you already know start function of a component should return a component map. This component basically applies function `inc` on any incoming value from its input and put it to its output.

② Gets the input of the current component

③ Gets the output of the current component

④ Connects input to output via an anonymous function which applies `inc` function to incoming values and dispatches them to the output

⑤ Simply consumes any incoming value and applies a function to that value

⑥ Defining all three components needed for this system to work. Please notice that we didn't defined any dependencies for these components. Pluging them to each other using a workflow catalog is a different story from component dependencies. We only need to define a component as a dependency if the second component use the first one directly in its start or stop function.

⑦ Defines a system with a linear workflow. In this case **HellHound** starts all the components in the system and then wires up components IO based on the desciption given by the `:workflow` key of the system.

⑧ Gets a component with the name from the default system.

⑨ Converts the vector to a stream source

⑩ Connects the stream source to the input of component1

In the system of this example, the workflow is like:

```
DATA ---> component-1 ---> component-2 ---> component-3
```

Component 3 don't have any output stream. But it can have one.

When we execute the above code the output would be something like:

The output of the above namespace

```
[17-10-29 21:52:37] [DEBUG] <hellhound.component:115> - Starting component ':simple-
system/component-2'...
[17-10-29 21:52:37] [DEBUG] <hellhound.component:115> - Starting component ':simple-
system/component-1'...
[17-10-29 21:52:37] [DEBUG] <hellhound.component:115> - Starting component ':simple-
system/component-3'...
[17-10-29 21:52:37] [INFO] <hellhound.system.core:142> - System started successfully.
[17-10-29 21:52:37] [DEBUG] <hellhound.system.workflow:107> - Setting up workflow...
[17-10-29 21:52:37] [INFO] <hellhound.system.workflow:110> - Workflow setup done.
Message Component 1:  1
Message Component 2:  2
Message Component 3:  3
Message Component 1:  2
Message Component 2:  3
Message Component 3:  4
Message Component 1:  3
Message Component 2:  4
Message Component 3:  5
Message Component 1:  4
Message Component 2:  5
Message Component 3:  6
Message Component 1:  5
Message Component 2:  6
Message Component 3:  7
Message Component 1:  6
Message Component 2:  7
Message Component 3:  8
Done.
```

As you can see the output above data flows to the pipeline and each component transoforms the value it gets from its input to a new value and put it into the output.

In the above example we used a simple vector and converted it to a stream source, but in practice the data flows to our system from an external source like a web server.

3.4.3. Conditional Workflow

This example is dedicated to conditional workflow. A quick demonstration of how you can setup a conditional workflow in your system.

In order to execute the following example from the hellhound_examples repository. Just issue the following command in the repository root directory.

```
$ lein run -m systems.simple-system3/main
```

Ok let’s jump into the code.

```
(ns systems.simple-system3
  (:require
   [manifold.stream :as s]
   [hellhound.system :as system]
   [hellhound.component :as hcomp]))

(defn start-fn1
  [component context]
  (let [input  (hcomp/input component)
        output (hcomp/output component)]
    (s/connect input output)) ①
  component)

(defn stop-fn
  [component]
  component)

(defn start-fn2
  [component context]
  (let [input (hcomp/input component)]
    (s/consume #(println "Odd: " %) input) ②
    component))

;; Start function of component-2.
(defn start-fn3
  [component context]
  (let [input (hcomp/input component)]
    (s/consume #(println "Even: " %) input) ③
    component))

(def component-1 (hcomp/make-component :simple-system/component-1 start-fn1 stop-fn))
(def component-2 (hcomp/make-component :simple-system/component-2 start-fn2 stop-fn))
(def component-3 (hcomp/make-component :simple-system/component-3 start-fn3 stop-fn))

(def simple-system
  {:components [component-2 component-1 component-3]
   :workflow [[:simple-system/component-1 odd? :simple-system/component-2] ④
              [:simple-system/component-1 even? :simple-system/component-3]]}) ⑤

(defn main
  []
  (system/set-system! simple-system)
  (system/start!)

  ;; Gets a component with the name from the default system.
  (let [component1 (system/get-component :simple-system/component-1)
        input      (hcomp/input component1)]

    (-> [1 2 3 4 5 6]
        (s/->source)
        (s/connect input))))
    (println "Done."))
```

① Connects the input of the component to its output with no transformation.

② Consumes values from the input and print them out.

③ Consumes values from the input and print them out.

④ Connects the output of `:simple-system/component-1` to `:simple-system/component-2`, But only delivers odd (hence the `odd?` predicate function) numbers to downstream component.

⑤ Connects the output of `:simple-system/component-1` to `:simple-system/component-3`, But only delivers even (hence the `even?` predicate function) numbers to downstream component.

In the system of this example, the workflow is like:

```
                  odd? | ---> component-2
DATA ---> component-1 |
                 even? | ---> component-3
```

When we execute the above code the output would be something like:

The output of the above namespace

```
[17-10-29 22:39:48] [DEBUG] <hellhound.component:115> - Starting component ':simple-
system/component-2'...
[17-10-29 22:39:48] [DEBUG] <hellhound.component:115> - Starting component ':simple-
system/component-1'...
[17-10-29 22:39:48] [DEBUG] <hellhound.component:115> - Starting component ':simple-
system/component-3'...
[17-10-29 22:39:48] [INFO] <hellhound.system.core:142> - System started successfully.
[17-10-29 22:39:48] [DEBUG] <hellhound.system.workflow:107> - Setting up workflow...
[17-10-29 22:39:48] [INFO] <hellhound.system.workflow:110> - Workflow setup done.
Odd:  1
Even:  2
Odd:  3
Even:  4
Odd:  5
Even:  6
Done.
```

The above output is clear enough. Component 2 got only odd values and component 3 got the even values. The predicate functions of a workflow catalog can be any function. But you need to bear in mind that **HellHound** is going to run these predicates for each value so for the sake of performance we need to keep the fast and pure.

Chapter 4. Buil-in Components

<PLACEHOLDER TEXT>

Chapter 5. HTTP

This chapter is dedicated to those features of **HellHound** which are related to Web. We'll learn all the necessary things to create a web application using **HellHound**.

5.1. Overview

Let's take tour around the HellHound's HTTP to see how it works on top of the HellHound's .

5.1.1. Installation

<PLACEHOLDER TEXT>

5.1.2. Building blocks

The HTTP package of **HellHound** is made from several smaller parts which work together. Parts such as HTTP router, Websocket server, Event Dispatcher and so on.

In general sense the HellHound's HTTP stack is just a barbone Ring compatible Pedestal webserver. The recommended way of using it is through `hellhound.components.webserver/factory` function with no parameter which creates a webserver with all the default values which in most cases you don't need to chaing. By default HellHound will spin up a webserver which only serves assets, root enpoint and a websocket endpoint and the rest of the process happens on the client side application (Which handles by HellHound again), So most of the time when you need a new route in your application, you have to add it on the client side router. Ofcourse you can choose to not follow the same pattern but then you won't get enough benefit of using **HellHound**.

HTTP package of **HellHound** has been composed by several pieces. In its heart it uses the Pedestal's Interceptors to reply to HTTP requests. So almost all of the Pedestal concepts applies to **HellHound** HTTP as well.

Another key part of HellHound HTTP package is the HTTP router. There are plenty or Clojure routers around but in my opinion the best of is the Pdestal Router. It's a data driven HTTP router with is simple, and predictable. We'll discuss Pedestal router later in this chapter.

The most important part of HTTP package is the webserver component which lives in `hellhound.components.webserver`. Under the hood it uses Aleph that is an asynchronous network library on top Netty.

5.1.3. Recommended solution

<PLACEHOLDER TEXT>

5.2. HTTP Router

If you’re using the default configuration of HellHound HTTP package (`hellhound.components.webserver/default-factory`). then you don’t need to learn about the router and can safely skip this section.,

This section contains Pedestal’s router reference which modified a little bit based on **HellHound** requirements.

HellHound uses `Pedestal Router` for HTTP routing, So everything you read in this section is applicable to your HellHound application as well.

5.2.1. Routes vs. Routers

HellHound distinguishes between routes and routers.

"Routes" are data that describe a structure of decisions to be made in handling requests.

"Routers" are the functions (supplied as Interceptors) that analyze incoming requests against the routes.

Pedestal uses protocols to connect routes to routers. Either can be varied independently.

5.2.2. Routes

Routes are data used by routers. The `verbose syntax` is the form used directly by routers. The `table` and `terse` syntaxes are convenient forms that can be expanded into the verbose syntax.

Users are free to describe routes however they like, so long as the data that reaches the router is in the `verbose syntax`.

The built in syntaxes are expanded by expand-routes polymorphically on the argument type:

Argument to `expand-routes`	**Syntax used**
Vector	Terse
Set	Table
Map	Verbose

`hellhound.http.route/expand-routes` is an alias for `io.pedestal.http.route/expand-routes`

5.2.3. Verbose Syntax

The verbose syntax is a list of maps, with the following structure.

```
  {:route-name :new-user
   :app-name   :example-app        ; optional
   :path       "/user/:id/*blah"   ; like Ruby on Rails
                                   ; (catch-all route is "/*path")
   :method     :post               ; or :any, :get, :put, ...
   :scheme     :https              ; optional
   :host       "example.com"       ; optional
   :port       "8080"              ; optional
   :interceptors [...]             ; vector of interceptors to be enqueued on the
context

   ;; Generated for path-matching:
   :path-re           #"/\Quser\E/([^/]+)/(.+)"
   :path-parts        ["user" :id :blah]
   :path-params       [:id :blah]
   :path-constraints  {:id "([^/]+)"
                       :blah "(.+)"}
   :query-constraints {:name #".+"
                       :search #"[0-9]+"}
   }
```

`:route-name` must be unique.

The keys `:path-re`, `:path-parts`, `:path-params`, and `:path-constraints` are derived from the `:path`.

Users will not generally write routes directly in verbose format.

5.2.4. Table Syntax

Table syntax is expanded by `expand-routes` into the full (verbose) syntax shown above.

When the argument to `expand-routes` is a **set**, it will be expanded using the table syntax.

In table syntax, each route is a vector of:

1. Path string
2. Verb. One of :any, :get, :put, :post, :delete, :patch, :options, :head
3. Handler or vector of interceptors
4. Optional route name clause
5. Optional constraint clause

example.clj

```
  ["/user/:id/*blah"  :post  [...] :route-name :new-user :constraints {:id #"[0-9]+"}]
```

The `:host`, `:port`, `:app-name`, and `:scheme` are provided in a map that applies to all routes in the list.

example.clj

```
(route/expand-routes
  #{{:host "example.com" :scheme :https}
    ["/user/:id/*blah"  :post  [...] :route-name :new-user :constraints {:id #"[0-9]+
"}]})
```

When multiple routes use the same path, they must differ by both verb and route name.

example.clj

```
    ["/user/:id"  :post new-user  :route-name :new-user  :constraints {:id #"[0-9]+"}]
    ["/user/:id"  :get  view-user :route-name :view-user :constraints {:id #"[0-9]+"}]
```

If the last interceptor in the chain has a name, or you supply a symbol that resolves to a function or named interceptor, then the route name will be derived from that.

example.clj

```
    ;; Route names will be taken from the symbols
    (defn new-user [request] ,,,)
    (defn view-user [request] ,,,)

    (route/expand-routes
      #{["/user/:id"  :post [,,, `new-user]  :constraints {:id #"[0-9]+"}]
        ["/user/:id"  :get  [,,, `view-user] :constraints {:id #"[0-9]+"}]})

    ;; Route names will be taken from the interceptors
    (def new-user-intc {:name :new-user :enter (fn [context] ,,,)})
    (def view-user-intc {:name :view-user :enter (fn [context] ,,,)})

    (route/expand-routes
      #{["/user/:id"  :post [,,, new-user-intc]  :constraints {:id #"[0-9]+"}]
        ["/user/:id"  :get  [,,, view-user-intc] :constraints {:id #"[0-9]+"}]})
```

5.2.5. Terse Syntax

Terse syntax is expanded by `expand-routes` into the full (verbose) syntax shown above.

When the argument to `expand-routes` is a **vector**, it will be expanded using the terse syntax.

In the terse format, a route table is a vector of nested vectors. Each top-level vector describes an "application". The application vector contain the following elements:

- (Optional) A keyword identifying the application by name
- (Optional) A URL scheme
- (Optional) A host name
- One or more nested vectors specifying routes

example.clj

```
;; Application vector with one route vector (which has one route)
[[:hello-world :http "example.com"
 ["/hello-world" {:get hello-world}]]]
```

Route vectors can be nested arbitrarily deep. Each vector adds a path segment. The nesting structure of the route vectors maps to the hierarchic tree structure of the routes.

Each route vector contains the following:

1. A path segment. This must begin with a slash.
2. (Optional) Interceptor metadata for the verb map.
3. (Optional) Constraint metadata for the verb map.
4. A verb map
5. Zero or more child route vectors

The allowed keys in a verb map are:

- :get
- :put
- :post
- :delete
- :any

The value of each key is either a handler function or a list of interceptors.

Each verb in the verb map defines a route on the path. This example defines four routes.

example.clj

```
[[:hello-world :http "example.com"
 ["/order" {:get list-orders
            :post create-order}
   ["/:id" {:get view-order
            :put update-order}]]]
```

Interceptor metadata applies to every route in the verb map. In this example `load-order-from-db` applies to both the `:get` and `:put` routes for the path "/order/:id"

example.clj

```clojure
[[:hello-world :http "example.com"
 ["/order" {:get list-orders
            :post create-order}
   ["/:id" ^{:interceptors [load-order-from-db]
             :constraints  {:id #"[0-9]+"}}
           {:get view-order
            :put update-order}]]]
```

(Recall that metadata is attached to the *next* data structure read. The metadata with constraints and interceptors will be attached to the verb map.)

If multiple routes have the same handler, you will need to distinguish them by providing route names. (This is necessary so URL generation knows which route to use.) A route name is a keyword that goes in the first position of an interceptor vector in the verb map. In the following example, both POST routes have the same handler. We provide the keywords `:post-without-id` and `:post-by-id` to distinguish the routes.

example.clj

```clojure
[[:hello-world :http "example.com"
 ["/order" {:get  list-orders
            :post [:post-without-id create-order]}
   ["/:id" {:get  query-order
            :post [:post-by-id create-order]}]]]
```

5.3. Routers

When your application starts a Pedestal service with `create-servlet` or `create-server`, Pedestal creates a router, using the following keys from the service map:

Key	Meaning
`:io.pedestal.http/routes`	Routes as described above
`:io.pedestal.http/router`	Key to select a router, or a function that constructs a router

When the value of `:io.pedestal.http/router` is a keyword, it selects one of the built in routers:

- `:map-tree`
- `:prefix-tree`
- `:linear-search`

Router	Performance	Scaling in # Routes	Limitations
Map Tree	Very fast	Constant	Applies when all routes are static. Falls back to prefix tree if any routes have path parameters or wildcards.
Prefix Tree	High performance, space efficient	Log32(N)	Wildcard routes always win over explicit paths in the same subtree. E.g., `/path/:wild` will always match, even if `/path/user` is defined
Linear Search	Lowest performance	O(N)	Routes are checked in order. Precedence is precise.

5.3.1. Custom Router

When the value of `:io.pedestal.http/router` is a function, that function is used to construct a router. The function must take one argument: the collection of fully expanded routes. It must return something that satisfies the `Router` protocol.

5.4. Examples

Now it's time to see HTTP package in action. Let's start from simple to hard.

5.4.1. Echo Server

In this example we're going to create a really simple system which contains a hellhound webserver which reponds with what is gets via the ws connection.

As you already know, HellHound communications happens over websocket connection which creates by the `hellhound.components.webserver` component.

In order to execute the following example from the hellhound_examples repository. Just issue the following command in the repository root directory.

```
$ lein run -m components.webserver-example1/main
```

Ok let's jump into the code.

```
(ns components.webserver-example1
  (:require
   [hellhound.system :as system]
   [hellhound.component :as hcomp]
   [hellhound.components.webserver :as web] ①
   [hellhound.components.transform :as transform] ②
   [hellhound.http :as http]
   [manifold.stream :as s]))

;; System definition.
(def system
  {:components
   [(web/factory http/default-routes) ③
    (transform/factory ::output ④
                       (fn [context msg]
                         (println "RECEIVED: " msg)
                         msg))]

   :workflow [[::web/webserver  ::output]
              [::output ::web/webserver]]}) ⑤

(defn main
  []
  (system/set-system! system)
  (system/start!))
```

① HellHound’s webserver component ns.

② HellHound’s transform component ns.

③ Webserver component. The name of this component would be `::hellhound.components.webserver/webserver` We used `hellhound.http/default-routes` as the routes definition. But you can provide your own routes as long as it contains the hellhound websocket endpoint. Also as a shortcut you can use `hellhound.components.webserver/default-factory`. It’s just a shortcut around `hellhound.components.webserver/factory`.

④ Transform component is a very simple component which redirects incoming messages from input stream to output stream and applies the given function to each message. In this case we don’t do any transformation. We just log the message.

⑤ A closed workflow. In this workflow the output of `::output` component would be the input of `::web/webserver` component while the output of `::web/webserver` would be the input of `::output`. The important thing to remember in this example is that basically the output of the webserver component is a stream of data which receives from clients via websocket. other requests to none websocket endpoint handle synchronously by pedestal interceptors. In general we highly recommend to avoid this situation and implement your views in client side which is connected to HellHound server by a websocket

Run the example code and then use below code in your browser console to test the echo server.

```
ws = new WebSocket("ws://localhost:3000/ws")
ws.onmessage = function(e) { console.log(e.data) }
ws.send("hello world")
```

After running the above code you’re going to see "hello world" on your js console.

In this example data flow from webserver component to the output component and back to webserver component as the input and webserver component send its input to the client side application using the ws connection.

Appendix A: Prerequisite

Before learning more about **HellHound** and how to use it, you need to know about several concepts and libraries in advance which are heavily used in **HellHound**. If you already know about tools and concepts such as Manifold, Deffered and Stream just skip to Getting Started section, Otherwise please read this section first.

A.1. Commander Pattern

Have you ever hit a wall with REST? Does modeling your problem-domain into CRUD-able entities feel like fitting a square peg into a round hole? Perhaps, instead of modeling our services like little databases, we should instead model them like reactors over immutable event streams.

Commander pattern is an application architecture design pattern. In order to know more about the concept, it is highly recommend to watch this video.

HellHound heavily inspired from this pattern.

A.2. Manifold

Most of the documentation about Manifold library have been taken and from it’s official documentation and modified to make the life of the user easier. Kudos to the authors of Manifold.

This library provides basic building blocks for asynchronous programming, and can be used as a translation layer between libraries which use similar but incompatible abstractions. Manifold provides two core abstractions: **deferreds**, which represent a single asynchronous value, and **streams**, which represent an ordered sequence of asynchronous values.

HellHound uses manifolds almost everywhere, so it’s a good idea learn about this fantastic library which is brought to us by @ztellman and the awesome contributors of this library.

Long story short, **Manifold** library provides awesome asynchronous values by **deferreds** and a super useful abstraction for **streams**.

If you’re interested in Manifold, you might want to know about the rationale behind it.

A.3. Deferred

A deferred in Manifold is similar to a Clojure promise:

```
> (require '[manifold.deferred :as d])
nil

> (def d (d/deferred))
#'d

> (d/success! d :foo)
true

> @d
:foo
```

However, similar to Clojure's futures, deferreds in Manifold can also represent errors. Crucially, they also allow for callbacks to be registered, rather than simply blocking on dereferencing.

```
> (def d (d/deferred))
#'d

> (d/error! d (Exception. "boom"))
true

> @d
Exception: boom
```

```
> (def d (d/deferred))
#'d

> (d/on-realized d
    (fn [x] (println "success!" x))
    (fn [x] (println "error!" x)))
<< ... >>

> (d/success! d :foo)
success! :foo
true
```

A.3.1. composing with deferreds

Callbacks are a useful building block, but they're a painful way to create asynchronous workflows. In practice, no one should ever use `on-realized`.

Instead, they should use `manifold.deferred/chain`, which chains together callbacks, left to right:

```
> (def d (d/deferred))
#'d

> (d/chain d inc inc inc #(println "x + 3 =" %))
<< ... >>

> (d/success! d 0)
x + 3 = 3
true
```

`chain` returns a deferred representing the return value of the right-most callback. If any of the functions returns a deferred or a value that can be coerced into a deferred, the chain will be paused until the deferred yields a value.

Values that can be coerced into a deferred include Clojure futures, Java futures, and Clojure promises.

```
> (def d (d/deferred))
#'d

> (d/chain d
    #(future (inc %))
    #(println "the future returned" %))
<< ... >>

> (d/success! d 0)
the future returned 1
true
```

If any stage in `chain` throws an exception or returns a deferred that yields an error, all subsequent stages are skipped, and the deferred returned by `chain` yields that same error. To handle these cases, you can use `manifold.deferred/catch`:

```
> (def d (d/deferred))
#p

> (-> d
    (d/chain dec #(/ 1 %))
    (d/catch Exception #(println "whoops, that didn't work:" %)))
<< ... >>

> (d/success! d 1)
whoops, that didn't work: #error {:cause Divide by zero :via [{:type
java.lang.ArithmeticException ...
true
```

Using the → threading operator, `chain` and `catch` can be easily and arbitrarily composed.

To combine multiple deferrable values into a single deferred that yields all their results, we can use `manifold.deferred/zip`:

```
> @(d/zip (future 1) (future 2) (future 3))
(1 2 3)
```

Finally, we can use `manifold.deferred/timeout!` to register a timeout on the deferred which will yield either a specified timeout value or a `TimeoutException` if the deferred is not realized within `n` milliseconds.

```
> @(d/timeout!
     (d/future (Thread/sleep 1000) :foo)
     100
     :bar)
:bar
```

Note that if a timeout is placed on a deferred returned by `chain`, the timeout elapsing will prevent any further stages from being executed.

`future` vs `manifold.deferred/future`

Clojure's futures can be treated as deferreds, as can Clojure's promises. However, since both of these abstractions use a blocking dereference, in order for Manifold to treat it as an asynchronous deferred value it must allocate a thread.

Wherever possible, use `manifold.deferred/deferred` instead of `promise`, and `manifold.deferred/future` instead of `future`. They will behave identically to their Clojure counterparts (`deliver` can be used on a Manifold deferred, for instance), but allow for callbacks to be registered, so no additional threads are required.

A.3.2. let-flow

Let's say that we have two services which provide us numbers, and want to get their sum. By using `zip` and `chain` together, this is relatively straightforward:

```
(defn deferred-sum []
  (let [a (call-service-a)
        b (call-service-b)]
    (chain (zip a b)
      (fn [[a b]]
        (+ a b)))))
```

However, this isn't a very direct expression of what we're doing. For more complex relationships between deferred values, our code will become even more difficult to understand. In these cases, it's often best to use `let-flow`.

```
(defn deferred-sum []
  (let-flow [a (call-service-a)
             b (call-service-b)]
    (+ a b)))
```

In `let-flow`, we can treat deferred values as if they're realized. This is only true of values declared within or closed over by `let-flow`, however. So we can do this:

```
(let [a (future 1)]
  (let-flow [b (future (+ a 1))
             c (+ b 1)]
    (+ c 1)))
```

but not this:

```
(let-flow [a (future 1)
           b (let [c (future 1)]
                (+ a c))]
  (+ b 1))
```

In this example, `c` is declared within a normal `let` binding, and as such we can't treat it as if it were realized.

It can be helpful to think of `let-flow` as similar to Prismatic's Graph library, except that the dependencies between values are inferred from the code, rather than explicitly specified. Comparisons to core.async's goroutines are less accurate, since `let-flow` allows for concurrent execution of independent paths within the bindings, whereas operations within a goroutine are inherently sequential.

`manifold.deferred/loop`

Manifold also provides a `loop` macro, which allows for asynchronous loops to be defined. Consider `manifold.stream/consume`, which allows a function to be invoked with each new message from a stream. We can implement similar behavior like so:

```
(require
  '[manifold.deferred :as d]
  '[manifold.stream :as s])

(defn my-consume [f stream]
  (d/loop []
    (d/chain (s/take! stream ::drained)

      ;; if we got a message, run it through `f`
      (fn [msg]
        (if (identical? ::drained msg)
          ::drained
          (f msg)))

      ;; wait for the result from `f` to be realized, and
      ;; recur, unless the stream is already drained
      (fn [result]
        (when-not (identical? ::drained result)
          (d/recur))))))
```

Here we define a loop which takes messages one at a time from `stream`, and passes them into `f`. If `f` returns an unrealized value, the loop will pause until it’s realized. To recur, we make sure the value returned from the final stage is `(manifold.deferred/recur & args)`, which will cause the loop to begin again from the top.

While Manifold doesn’t provide anything as general purpose as core.async’s `go` macro, the combination of `loop` and `let-flow` can allow for the specification of highly intricate asynchronous workflows.

A.3.3. custom execution models

Both deferreds and streams allow for custom execution models to be specified. To learn more, //aleph.io/docs/execution.md[go here].

A.4. Stream

A Manifold stream can be created using `manifold.stream/stream`:

```
> (require '[manifold.stream :as s])
nil
> (def s (s/stream))
#'s
```

A stream can be thought of as two separate halves: a **sink** which consumes messages, and a **source** which produces them. We can `put!` messages into the sink, and `take!` them from the source:

```
> (s/put! s 1)
<< ... >>
> (s/take! s)
<< 1 >>
```

Notice that both put! and take! return //aleph.io/manifold/deferred.md[deferred values]. The deferred returned by put! will yield true if the message was accepted by the stream, and false otherwise; the deferred returned by take! will yield the message.

Sinks can be **closed** by calling close!, which means they will no longer accept messages.

```
> (s/close! s)
nil
> @(s/put! s 1)
false
```

We can check if a sink is closed by calling closed?, and register a no-arg callback using on-closed to be notified when the sink is closed.

Sources that will never produce any more messages (often because the corresponding sink is closed) are said to be **drained**. We may check whether a source is drained via drained? and on-drained.

By default, calling take! on a drained source will yield a message of nil. However, if nil is a valid message, we may want to specify some other return value to denote that the source is drained:

```
> @(s/take! s ::drained)
::drained
```

We may also want to put a time limit on how long we're willing to wait on our put or take to complete. For this, we can use try-put! and try-take!:

```
> (def s (s/stream))
#'s
> @(s/try-put! s :foo 1000 ::timeout)
::timeout
```

Here we try to put a message into the stream, but since there are no consumers, it will fail after waiting for 1000ms. Here we've specified ::timeout as our special timeout value, otherwise, it would simply return false.

```
> @(s/try-take! s ::drained 1000 ::timeout)
::timeout
```

Again, we specify the timeout and special timeout value. When using try-take!, we must specify return values for both the drained and timeout outcomes.

A.4.1. stream operators

The simplest thing we can do a stream is consumed every message that comes into it:

```
> (s/consume #(prn 'message! %) s)
nil
> @(s/put! s 1)
message! 1
true
```

However, we can also create derivative streams using operators analogous to Clojure's sequence operators, a full list of which [can be found here](http://ideolalia.com/manifold/):

```
> (->> [1 2 3]
    s/->source
    (s/map inc)
    s/stream->seq)
(2 3 4)
```

Here, we've mapped `inc` over a stream, transforming from a sequence to a stream and then back to a sequence for the sake of a concise example. Note that calling `manifold.stream/map` on a sequence will automatically call `→source`, so we can actually omit that, leaving just:

```
> (->> [1 2 3]
    (s/map inc)
    s/stream->seq)
(2 3 4)
```

Since streams are not immutable, in order to treat it as a sequence we must do an explicit transformation via `stream→seq`:

```
> (->> [1 2 3]
    s/->source
    s/stream->seq
    (map inc))
(2 3 4)
```

Note that we can create multiple derived streams from the same source:

```
> (def s (s/stream))
#'s
> (def a (s/map inc s))
#'a
> (def b (s/map dec s))
#'b
> @(s/put! s 0)
true
> @(s/take! a)
1
> @(s/take! b)
-1
```

Here, we create a source stream `s`, and map `inc` and `dec` over it. When we put our message into `s` it immediately is accepted, since `a` and `b` are downstream. All messages put into `s` will be propagated into **both** `a` and `b`.

If `s` is closed, both `a` and `b` will be closed, as will any other downstream sources we've created. Likewise, if everything downstream of `s` is closed, `s` will also be closed. This is almost always desirable, as failing to do this will simply cause `s` to exert back pressure on everything upstream of it. However, If we wish to avoid this behavior, we can create a `(permanent-stream)`, which cannot be closed.

For any Clojure operation that doesn't have an equivalent in `manifold.stream`, we can use `manifold.stream/transform` with a transducer:

```
> (->> [1 2 3]
    (s/transform (map inc))
    s/stream->seq)
(2 3 4)
```

There's also `(periodically period f)`, which behaves like `(repeatedly f)`, but will emit the result of `(f)` every `period` milliseconds.

A.4.2. connecting streams

Having created an event source through composition of operators, we will often want to feed all messages into a sink. This can be accomplished via `connect`:

```
> (def a (s/stream))
#'a
> (def b (s/stream))
#'b
> (s/connect a b)
true
> @(s/put! a 1)
true
> @(s/take! b)
1
```

Again, we see that our message is immediately accepted into `a`, and can be read from `b`. We may also pass an options map into `connect`, with any of the following keys:

Field	Description
`downstream?`	whether the source closing will close the sink, defaults to `true`
`upstream?`	whether the sink closing will close the source, **even if there are other sinks downstream of the source**, defaults to `false`
`timeout`	the maximum time that will be spent waiting to convey a message into the sink before the connection is severed, defaults to `nil`
`description`	a description of the connection between the source and sink, useful for introspection purposes

Upon connecting two streams, we can inspect any of the streams using `description`, and follow the flow of data using `downstream`:

```
> (def a (s/stream))
#'a
> (def b (s/stream))
#'b
> (s/connect a b {:description "a connection"})
nil
> (s/description a)
{:pending-puts 0, :drained? false, :buffer-size 0, :permanent? false, ...}
> (s/downstream a)
(["a connection" << stream: ... >>])
```

We can recursively apply `downstream` to traverse the entire topology of our streams. This can be a powerful way to reason about the structure of our running processes, but sometimes we want to change the message from the source before it's placed into the sink. For this, we can use `connect-via`:

```
> (def a (s/stream))
#'a
> (def b (s/stream))
#'b
> (s/connect-via a #(s/put! b (inc %)) b)
nil
```

Note that `connect-via` takes an argument between the source and sink, which is a single-argument callback. This callback will be invoked with messages from the source, under the assumption that they will be propagated to the sink. This is the underlying mechanism for `map`, `filter`, and other stream operators; it allows us to create complex operations that are visible via `downstream`:

```
> (def a (s/stream))
#'a
> (s/map inc a)
<< source: ... >>
> (s/downstream a)
([{:op "map"} << sink: {:type "callback"} >>])
```

Each element returned by `downstream` is a 2-tuple, the first element describing the connection, and the second element describing the stream it's feeding into.

The value returned by the callback for `connect-via` provides backpressure - if a deferred value is returned, further messages will not be passed in until the deferred value is realized.

A.4.3. buffers and backpressure

We saw above that if we attempt to put a message into a stream, it won't succeed until the value is taken out. This is because the default stream has no buffer; it simply conveys messages from producers to consumers. If we want to create a stream with a buffer, we can simply call `(stream buffer-size)`. We can also call `(buffer size stream)` to create a buffer downstream of an existing stream.

We may also call `(buffer metric limit stream)`, if we don't want to measure our buffer's size in messages. If, for instance, each message is a collection, we could use `count` as our metric, and set `limit` to whatever we want the maximum aggregate count to be.

To limit the rate of messages from a stream, we can use `(throttle max-rate stream)`.

A.4.4. event buses and publish/subscribe models

Manifold provides a simple publish/subscribe mechanism in the `manifold.bus` namespace. To create an event bus, we can use `(event-bus)`. To publish to a particular topic on that bus, we use `(publish! bus topic msg)`. To get a stream representing all messages on a topic, we can call `(subscribe bus topic)`.

Calls to `publish!` will return a deferred that won't be realized until all streams have accepted the message. By default, all streams returned by `subscribe` are unbuffered, but we can change this by providing a `stream-generator` to `event-bus`, such as `(event-bus #(stream 1e3))`. A short example of how `event-bus` can be used in concert with the buffering and flow control mechanisms [can be found here](https://youtu.be/1bNOO3xxMc0?t=1887). n

Appendix B: Contributing to HellHound

Wow, thanks for your interest in helping out with HellHound. Let this document serve as your guide.

B.1. Looking for work?

If you're looking for a task to work on, check out the TODO in our issues.

Less defined tasks will be marked with the discuss label. Jump in here if you want to be a part of something big.

B.2. New Features

HellHound is the thinking persons framework, so every contribution starts with some **deeeep** thought. Finished?

Alright, your next step is to start a discussion.

Create an issue to start a conversation. Tell us what you're trying to accomplish and how you think you might do it. If all is well, we'll probably give you the :thumbsup: to start developing.

B.3. Bugs

Of course, if you run into any straight-up bugs or weirdness feel free to skip the thinking (or at least too much of it) and immediately submit an issue.

We have an issue template in place that will ask you some details about the platform you are running and how to reproduce the bug. (If you can reproduce it reliably. If not, go ahead and file the issue anyway so we can start looking at it.)

Some of the usual stuff we'll want to know:

- What happened?
  - "I manifested a being from the outer dimensions."
- What did you expect to happen?
  - "Hello, world."
- How can you reprodice it?
  - "I created a new HellHound service with the template, then installed some code that Bob Howard gave me."
- What operating system and version are you using?
  - e.g. "OS X 10.8"
- What version of Clojure, Java, and Leiningen or Boot are you using?
  - e.g. "Leiningen 2.5.2 on Java 1.8.0_u40 Java HotSpot™ 64-Bit Server VM"

- What HellHound version are you using?
  - e.g. "0.5.0"
  - or, for a SNAPSHOT: "0.5.1-SNAPSHOT at d0cf2b4"

Even better, include a link to a gist or repository where we can jump straight to the problem.

B.4. Tests

<PLACEHOLDER TEXT>

Appendix C: Contributor Covenant Code of Conduct

C.1. Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

C.2. Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

C.3. Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

C.4. Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

C.5. Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at devs@codamic.tech. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

C.6. Attribution

This Code of Conduct is adapted from the [Contributor Covenant][homepage], version 1.4, available at version

homepage version

Colophon

The HellHound User Manual

© 2017 by Sameer Rahmani <lxsameer@gnu.org>

Created in Asciidoctor, Debian Unstable and FG42 Editor.