Anatomy of a Real-Time Elixir App

Stephen Bussey

Hi, I'm Steve Bussey

- Software Architect at SalesLoft (Atlanta, GA, USA)
- Elixir in production for several years
- I love Elixir!

The Pragmatic Programmers

Real-Time Phoenix

Build Highly Scalable Systems with Channels

Stephen Bussey

Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

https://sal.es/rtp

Goals for Today

– – –

- Understand importance of reading the source code of libraries that we use
- Dive deep into the basics of how Phoenix Channels work

Real-Time Elixir Apps

What is a Real-Time App?

- Interact with clients and keep them up to date as data changes
- Goals set around how long it takes for data changes to propagate from source to client
- A real-time app should ideally always reflect the “truth”

Components of a Real-Time App

- Client (JavaScript, iPhone app, etc)
- Server-Client connection (ex. WebSocket)
- Server (Elixir, Phoenix)
- Data Pipeline (GenStage, homemade)

We're going to look at and break down the server component today. It would take awhile to do this for all of the different components!

Anatomy

What is anatomy?

- “A study of the structure or internal workings of something.”
- Concerned with the relationship between small parts of the whole

Anatomy of a Software Library

– – –

- We can get a clearer picture of the whole by understanding the parts
- We can more easily dig in and debug problems that pop up
- We gain self-sufficiency in our usage of a library

Self-sufficiency takes us to the next level

Self-sufficient developers

- Can answer questions that may not be immediately clear
- Can more easily contribute to open-source or internal codebases
- Get to the bottom of problems (bugs) to find a path forward

Our Path to Self-Sufficiency

– – –

- Approach a library with curiosity
- Dig in when a problem appears
- Contribute back any learnings, such as documentation or found defects

Phoenix Channels

Implementing the 101 of Channels

– – –

- Basic Socket
- Channel with a message handler
- JavaScript connects to the Socket and Channel
- Let’s view that code

Lots of moving parts, not much code

– – –

- What is a Socket, and how does it connect?
- How does a Channel differ from a Socket?
- How does a Channel become joined?
- How does a message get processed and responded to?

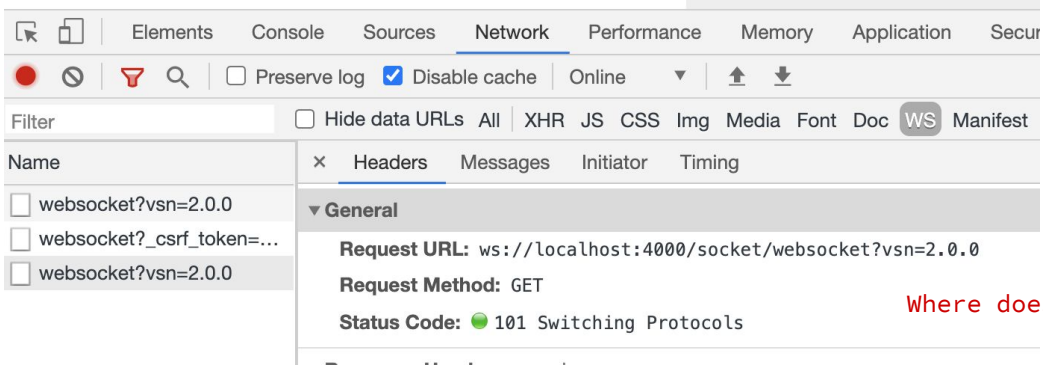
Let’s dig into the code

– – –

- We’ll only go deeper if we have a clear reason to (no magic jumps)
- Assume no prior knowledge, we have to see it ourselves
- We may end up with more questions than answers, but we’ll answer our key questions

Socket connection and routing

WebSocket is used to connect to the Socket



Where does this go?

Socket Definition

- Defined /socket in **DemoWeb.Endpoint**
- Leverages the socket/3 function of **Phoenix.Endpoint**

```
socket "/socket", DemoWeb.UserSocket,
  websocket: true,
  longpoll: false
```

Phoenix.Endpoint defmacro socket/3

```
848   defmacro socket(path, module, opts \\ []) do
849     # Tear the alias to simply store the root in the AST.
850     # This will make Elixir unable to track the dependency
851     # between endpoint <-> socket and avoid recompiling the
852     # endpoint (alongside the whole project ) whenever the
853     # socket changes.
854     module = tear_alias(module)
855
856     quote do
857       @phoenix_sockets {unquote(path), unquote(module), unquote(opts)}
858     end
859   end
```

How is phoenix_sockets used?

Phoenix.Endpoint before_compile

```
585   @doc false
586   defmacro __before_compile__(%{module: module}) do
587     sockets = Module.get_attribute(module, :phoenix_sockets)
588
589     dispatches =
590       for {path, socket, socket_opts} <- sockets,
591           {path, type, conn_ast, socket, opts} <- socket_paths(module, path, socket, socket_opts) do
592         quote do
593           defp do_handler(unquote(path), conn, _opts) do
594             {unquote(type), unquote(conn_ast), unquote(socket), unquote(Macro.escape(opts))}
595           end
596         end
597       end
```

Each dispatch becomes a quoted function

Phoenix.Endpoint dispatches

```
624         @doc false
625         def __handler__(%{path_info: path} = conn, opts), do: do_handler(path, conn, opts)
626         unquote(dispatches)
627         defp do_handler(_path, conn, opts), do: {:plug, conn, __MODULE__, opts}
```

Pretty cool

When Endpoint is called, it will execute the dispatch functions, instead of going into Plug

Phoenix.Endpoint socket_paths

```
631  defp socket_paths(endpoint, path, socket, opts) do
632    paths = []
633    websocket = Keyword.get(opts, :websocket, true)
634    longpoll = Keyword.get(opts, :longpoll, false)
635
636    paths =
637      if websocket do
638        config = Phoenix.Socket.Transport.load_config(websocket, Phoenix.Transports.WebSocket)
639        {conn_ast, match_path} = socket_path(path, config)
640        [{match_path, :websocket, conn_ast, socket, config} | paths]
641      else
642        paths
643      end
644
645    paths =
646      if longpoll do
647        config = Phoenix.Socket.Transport.load_config(longpoll, Phoenix.Transports.LongPoll)
```

A WebSocket is configured via Transports.WebSocket, which contributes to building up the dispatch AST

Output of do_handler

- Inspected by modifying dependency source code

What the heck? No function calls

```
{:defp, [context: Phoenix.Endpoint, import: Kernel],
 [
   {:do_handler, [context: Phoenix.Endpoint],
    [
      ["socket", "websocket"],
      {:conn, [], Phoenix.Endpoint},
      {:_opts, [], Phoenix.Endpoint}
    ]},
   [
     do: {:{}, [],
      [
        :websocket,
        {:conn, [], Phoenix.Endpoint},
        DemoWeb.UserSocket,
        [
          path: "/websocket",
          serializer: [
            {Phoenix.Socket.V1.JSONSerializer, "~> 1.0.0"},
            {Phoenix.Socket.V2.JSONSerializer, "~> 2.0.0"}
          ],
          timeout: 60000,
          transport_log: false,
          compress: false
        ]
      ]}
   ]
 ]}
```

This is what a handler AST looks like

Coming Together

- Neat metaprogramming techniques!
- Phoenix maps each defined socket into a set of dispatch functions
- Phoenix supports WebSocket and LongPoll
- But we’re seemingly stuck

Grep is curiosity's best friend

- __handler__ is very specific
- Only appears in endpoint.ex and a cowboy2_handler.ex

```
➜  demo grep -R "__handler__" deps/
deps//phoenix/lib/phoenix/endpoint/cowboy2_handler.ex:      case endpoint.__handler__(conn, opts) do
deps//phoenix/lib/phoenix/endpoint/cowboy2_handler.ex:          [{^endpoint, :__handler__, _, _} | _] when reason == :undef and retry? ->
deps//phoenix/lib/phoenix/endpoint.ex:      def __handler__(%{path_info: path} = conn, opts), do: do_handler(path, conn, opts)
➜  demo
```

Now we're unstuck

Cowboy2Handler init/4

Transports.Websocket controls connections

cowboy_websocket is the response

```
18    defp init(conn, endpoint, opts, retry?) do
19      try do
20        case endpoint.__handler__(conn, opts) do
21          {:websocket, conn, handler, opts} ->
22            case Phoenix.Transports.WebSocket.connect(conn, endpoint, handler, opts) do
23              {:ok, %Plug.Conn{adapter: {@connection, req}} = conn, state} ->
24                cowboy_opts =
25                  opts
26                  |> Enum.flat_map(fn
27                    {:timeout, timeout} -> [idle_timeout: timeout]
28                    {:compress, _} = opt -> [opt]
29                    {:max_frame_size, _} = opt -> [opt]
30                    _other -> []
31                  end)
32                  |> Map.new()
33
34                {:cowboy_websocket, copy_resp_headers(conn, req), [handler | state], cowboy_opts}
35
36              {:error, %Plug.Conn{adapter: {@connection, req}} = conn} ->
37                {:ok, copy_resp_headers(conn, req), {handler, opts}}
38            end
```

Each WebSocket is processed by the Transport.connect function and returned as a cowboy websocket

Fork in the road. We need to know about the Transport and also Cowboy

Transports.WebSocket

Hey, it's our Socket!
Sort of...

```elixir
def connect(%{method: "GET"} = conn, endpoint, handler, opts) do
  conn
  |> Plug.Conn.fetch_query_params()
  |> Transport.code_reload(endpoint, opts)
  |> Transport.transport_log(opts[:transport_log])
  |> Transport.force_ssl(handler, endpoint, opts)
  |> Transport.check_origin(handler, endpoint, opts)
  |> Transport.check_subprotocols(opts[:subprotocols])
  |> case do
    %{halted: true} = conn ->
      {:error, conn}

    %{params: params} = conn ->
      keys = Keyword.get(opts, :connect_info, [])
      connect_info = Transport.connect_info(conn, endpoint, keys)
      config = %{endpoint: endpoint, transport: :websocket, options: opts

      case handler.connect(config) do
        {:ok, state} -> {:ok, conn, state}
        :error -> {:error, Plug.Conn.send_resp(conn, 403, "")}
        {:error, reason} ->
          {m, f, args} = opts[:error_handler]
          {:error, apply(m, f, [conn, reason | args])}
      end
  end
end

def connect(conn, _, _, _) do
  {:error, Plug.Conn.send_resp(conn, 400, "")}
end

def handle_error(conn, _reason), do: Plug.Conn.send_resp(conn, 403, "")
```

The Transport is a functional pipeline that goes into setting up and handling the request

It is NOT a Plug

We can see that handler.connect is called, but our app defines /3 and not /1

cowboy_websocket

Description

The module `cowboy_websocket` implements Websocket as a Ranch protocol. It also defines a callback interface for handling Websocket connections.

Callbacks

Websocket handlers must implement the following callback interface:

```
init(Req, State)
    -> {cowboy_websocket, Req, State}
     | {cowboy_websocket, Req, State, Opts}

websocket_init(State)            -> CallResult  %% optional
websocket_handle(InFrame, State) -> CallResult
websocket_info(Info, State)      -> CallResult
```

At the end of the day, Phoenix wraps around Cowboy for its web functionality

Coming Together

– – –

- Phoenix uses *cowboy_websocket* to power it's WS implementation
- *Transports.WebSocket* doesn't go through the application router or plug, it's a function pipeline
- Our **AppSocket.connect/1** function is called by *Transports.WebSocket*
- What defines **connect/1?**

Phoenix.Socket connect/1

```
286       @doc false
287       def connect(map), do: Phoenix.Socket.__connect__(__MODULE__, map, @phoenix_socket_options)
```

__connect__/3

```
433       {:ok, serializer} ->
434         result = user_connect(user_socket, endpoint, transport, serializer, params, connect_info)
435
```

user_connect/6

```
552     connect_result =
553       if function_exported?(handler, :connect, 3) do
554         handler.connect(params, socket, connect_info)
555       else
556         handler.connect(params, socket)
557       end
558
559     case connect_result do
560       {:ok, %Socket{} = socket} ->
561         case handler.id(socket) do
562           nil ->
563             {:ok, {state, socket}}
564
565           id when is_binary(id) ->
566             {:ok, {state, %{socket | id: id}}}
```

Phoenix proxies the connection request through multiple layers until it finally gets to our AppSocket.connect function

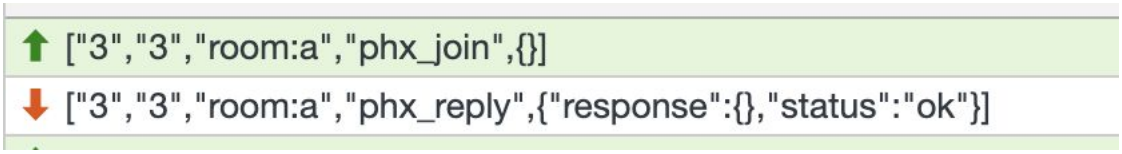
The result of handler.connect indicates whether the Socket will be allowed, or will fail

How does a Channel Join?

Establish our starting point

– – –

- Network Inspector for WebSocket shows a phx_join being sent to server
- Message format looks foreign (now)



Grep for websocket_handle, based on cowboy_websockets interface

```
→  demo grep -R "websocket_handle" deps/**/*.ex
deps/phoenix/lib/phoenix/endpoint/cowboy2_handler.ex:  def websocket_handle({opcode, payload}, [handler | state]) when opcode in [:text, :binary] do
deps/phoenix/lib/phoenix/endpoint/cowboy2_handler.ex:  def websocket_handle({opcode, payload}, handler_state) when opcode in [:ping, :pong] do
deps/phoenix/lib/phoenix/endpoint/cowboy2_handler.ex:  def websocket_handle(opcode, handler_state) when opcode in [:ping, :pong] do
deps/phoenix/lib/phoenix/endpoint/cowboy2_handler.ex:  def websocket_handle(_other, handler_state) do
```

Cowboy2Handler.websocket_handle/2

```
133    def websocket_handle({opcode, payload}, [handler | state]) when opcode in [:text, :binary] do
134      handle_reply(handler, handler.handle_in({payload, opcode: opcode}, state))
135    end
136
137    def websocket_handle({opcode, payload}, handler_state) when opcode in [:ping, :pong] do
138      handle_control_frame({payload, opcode: opcode}, handler_state)
139    end
140
141    def websocket_handle(opcode, handler_state) when opcode in [:ping, :pong] do
142      handle_control_frame({nil, opcode: opcode}, handler_state)
143    end
144
145    def websocket_handle(_other, handler_state) do
146      {:ok, handler_state}
147    end
```

It's our DemoSocket!

We know that cowboy is used to process websocket messages, so we know that is where we need to look next

Our DemoSocket.handle_in function is called for text or binary messages

Phoenix.Socket handle_in/2

```
292       @doc false
293       def handle_in(message, state), do: Phoenix.Socket.__in__(message, state)
```

__in__/2

```
466   def __in__({payload, opts}, {state, socket}) do
467     %{topic: topic} = message = socket.serializer.decode!(payload, opts)
468     handle_in(Map.get(state.channels, topic), message, state, socket)
469   end
```

Wait a second...is this a Channel<->topic mapping?

Socket.handle_in phx_join

```elixir
599  defp handle_in(nil, %{event: "phx_join", topic: topic, ref: ref, join_ref: join_ref} = message, state, socket) do
600    case socket.handler.__channel__(topic) do
601      {channel, opts} ->
602        case Phoenix.Channel.Server.join(socket, channel, message, opts) do
603          {:ok, reply, pid} ->
604            reply = %Reply{join_ref: join_ref, ref: ref, topic: topic, status: :ok, payload: reply}
605            state = put_channel(state, pid, topic, join_ref)
606            {:reply, :ok, encode_reply(socket, reply), {state, socket}}
607
608          {:error, reply} ->
609            reply = %Reply{join_ref: join_ref, ref: ref, topic: topic, status: :error, payload: reply}
610            {:reply, :error, encode_reply(socket, reply), {state, socket}}
611        end
612
613      _ ->
614        {:reply, :error, encode_ignore(socket, message), {state, socket}}
615    end
616  end
```

We'll come to this in a minute

Note that arg 1 is `nil`. That is the channel_pid and makes sense because the channel hasn't been joined yet

What is Channel.Server? We'll see soon.

Notice lots of error handling. Each layer that we've gone through so far is usually wrapped in error handling

Socket.put_channel

```
650   defp put_channel(state, pid, topic, join_ref) do
651     %{channels: channels, channels_inverse: channels_inverse} = state
652     monitor_ref = Process.monitor(pid)
653
654     %{
655       state |
656         channels: Map.put(channels, topic, {pid, monitor_ref}),
657         channels_inverse: Map.put(channels_inverse, pid, {topic, join_ref})
658     }
659   end
```

Helps answer “What is a Channel?”

Understanding topics and Channels is one of the most core pieces of real-time phoenix

A topic is a key in a map. Used for routing

A Channel is a process that responds to messages from the client

Regroup

– – –

- Cowboy handler routes all incoming messages through **DemoSocket**
- Each Socket is a new Process
- Top-level functions handled by **use Phoenix.Socket**
- Channels are just an entry in a map
- Each Channel must be a process (can be monitored)

Back to Our Questions

- What is a Socket, and how does it connect?
- How does a Channel differ from a Socket?
- How does a Channel become joined?
- How does a message get processed and responded to?

Channel Startup

Channel.Server join/4

```elixir
def join(socket, channel, message, opts) do
  %{topic: topic, payload: payload, ref: join_ref} = message
  assigns = Map.merge(socket.assigns, Keyword.get(opts, :assigns, %{}))
  socket = %{socket | topic: topic, channel: channel, join_ref: join_ref, assigns: assigns}

  ref = make_ref()
  from = {self(), ref}
  child_spec = channel.child_spec({socket.endpoint, from})

  case PoolSupervisor.start_child(socket.endpoint, socket.handler, from, child_spec) do
    {:ok, pid} ->
      send(pid, {Phoenix.Channel, payload, from, socket})
      mon_ref = Process.monitor(pid)

      receive do
        {^ref, {:ok, reply}} ->
          Process.demonitor(mon_ref, [:flush])
          {:ok, reply, pid}
```

A GenServer is started and sent a message before the Channel is joined

A Supervisor creates a child process for a Channel.Server

The process is created BEFORE it’s attempted to be joined. This allows you to send(self()) messages in the join/3 function and it works as expected

Channel.Server handle_info initial

```
287  def handle_info({Phoenix.Channel, auth_payload, {pid, _} = from, socket}, ref) do
288    Process.demonitor(ref)
289    %{channel: channel, topic: topic, private: private} = socket
290    Process.put(:"$callers", [pid])
291
292    socket = %{
293      socket
294      | channel_pid: self(),
295        private: Map.merge(channel.__socket__(:private), private)
296    }
297
298    start = System.monotonic_time()
299    {reply, state} = channel_join(channel, topic, auth_payload, socket)
300    duration = System.monotonic_time() - start
301    metadata = %{params: auth_payload, socket: socket, result: elem(reply, 0)}
302    :telemetry.execute([:phoenix, :channel_joined], %{duration: duration}, metadata)
303    GenServer.reply(from, reply)
304    state
305  end
```

This function processes the initial request to join a Channel

Line 299 is invocation of the join function, everything else is setup or monitoring code

Channel.Server channel_join

```
376   defp channel_join(channel, topic, auth_payload, socket) do
377     case channel.join(topic, auth_payload, socket) do
378       {:ok, socket} ->
379         {{:ok, %{}}, init_join(socket, channel, topic)}
380
381       {:ok, reply, socket} ->
```

init_join

```
400 defp init_join(socket, channel, topic) do
401   %{transport_pid: transport_pid, serializer: serializer, pubsub_server: pubsub_server} = socket
402     ...
418   Process.monitor(transport_pid)
419   fastlane = {:fastlane, transport_pid, serializer, channel.__intercepts__()}
420   PubSub.subscribe(pubsub_server, topic, metadata: fastlane)
421
422   {:noreply, %{socket | joined: true}}
423 end
```

Hey, it's the PubSub subscription!

If you feel curious, you can dig into "fastlane" here. I typically would because it would be an unknown if it's the first time I've seen it

Channel.Server handle_info proxy

```
341   def handle_info(msg, %{channel: channel} = socket) do
342     if function_exported?(channel, :handle_info, 2) do
343       msg
344       |> socket.channel.handle_info(socket)
345       |> handle_result(:handle_info)
346     else
347       warn_unexpected_msg(:handle_info, 2, msg)
348       {:noreply, socket}
349     end
350   end
```

Our app's Channel is not actually a GenServer module, it's a proxy for one!

Very important to understand the Channel process structure. What are the processes (Channel.Server, cowboy_websocket handler), process facades (YourChannel), or handlers (YourSocket)?

Regroup: What is a Channel?

- GenServer backed by Phoenix.Channel.Server
- A Channel can receive any unhandled messages via handle_info
- Lives under a Phoenix.Socket, mapped by topic

Channel Incoming Message & Reply

Phoenix.Socket handle_in

```
629   defp handle_in({pid, _ref}, message, state, socket) do
630     send(pid, message)
631     {:ok, {state, socket}}
632   end
```

This looks pretty simple, it just sends a message to the channel process

What happens when a Socket gets a new message? It calls this function (or another, based on pattern-matching)

Channel.Server handle_info

```
311   def handle_info(
312         %Message{topic: topic, event: event, payload: payload, ref: ref},
313         %{topic: topic} = socket
314       ) do
315     start = System.monotonic_time()
316     result = socket.channel.handle_in(event, payload, put_in(socket.ref, ref))
317     duration = System.monotonic_time() - start
318     metadata = %{ref: ref, event: event, params: payload, socket: socket}
319     :telemetry.execute([:phoenix, :channel_handled_in], %{duration: duration}, metadata)
320     handle_in(result)
321   end
```

Again, pattern matching plays a big part here. There are other “admin” type messages to handle

A lot of code that we see in these files is a single important line surrounded by monitoring or error handling

Channel.Server handle_in result

Replies are handled based on the Channel response

```
487   ## Handle in/replies
488
489   defp handle_in({:reply, reply, %Socket{} = socket}) do
490     handle_reply(socket, reply)
491     {:noreply, put_in(socket.ref, nil)}
492   end
493
494   defp handle_in({:stop, reason, reply, socket}) do
495     handle_reply(socket, reply)
496     handle_result({:stop, reason, socket}, :handle_in)
497   end
498
499   defp handle_in(other) do
500     handle_result(other, :handle_in)
501   end
502
```

Pattern matching is leveraged to decide how to handle different reply formats

Channel.Server handle_reply

```
503    defp handle_reply(socket, {status, payload}) when is_atom(status) and is_map(payload) do
504      reply(
505        socket.transport_pid,
506        socket.join_ref,
507        socket.ref,
508        socket.topic,
509        {status, payload},
510        socket.serializer
511      )
512    end
```

transport_pid = cowboy_websocket handler process

reply

```
248    def reply(pid, join_ref, ref, topic, {status, payload}, serializer)
249        when is_binary(topic) and is_map(payload) do
250      reply = %Reply{topic: topic, join_ref: join_ref, ref: ref, status: status, payload: payload}
251      send(pid, serializer.encode!(reply))
252      :ok
253    end
254
```

A reply is a message being sent to a process (cowboy_websocket transport)

All messages have a particular format dictated by the Serializer

Socket.V2.JSONSerializer.encode!

```elixir
def encode!(%Reply{} = reply) do
  data = [
    reply.join_ref,
    reply.ref,
    reply.topic,
    "phx_reply",
    %{status: reply.status, response: reply.payload}
  ]

  {:socket_push, :text, Phoenix.json_library().encode_to_iodata!(data)}
end
```

Messages are converted to/from this array format

Regroup: How are Channel Messages Processed?

– – –

- Each message is handled by the Channel Server
- A given message is processed serially, one at a time
- The Socket transport is sent the serialized reply
- All messages over the wire (up/down) are serialized into a particular format

Back to Our Questions

- What is a Socket, and how does it connect?
- How does a Channel differ from a Socket?
- How does a Channel become joined?
- How does a message get processed and responded to?
- We have (rough) answers to everything!

Quick Dive Into LiveView (time permitting)

– – –

- Feels more familiar than unknown
- LiveView mounts a Phoenix.Socket module called **Phoenix.LiveView.Socket**
- **Phoenix.LiveView.Channel** follows the Phoenix.Channel behaviour contract
- It implements all of the important bits, but doesn't *use Phoenix.Channel*

The Phoenix real-time stack is incredibly powerful.

You write not a lot of application code, but get rich real-time features.

Read through the source code if you hit a snag, or are just curious.

You may unblock yourself or learn something new, but you'll definitely become more self-sufficient.

Thank You!

– – –

- Book: sal.es/rtp
- Twitter: @yoooodaaaa
- Email: steve@salesloft.com
- Slides with Notes: https://bit.ly/anatomy-real-time-elixir