Extracted from:

Real-Time Phoenix

Build Highly Scalable Systems with Channels

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Real-Time Phoenix

Build Highly Scalable Systems with Channels

Stephen Bussey

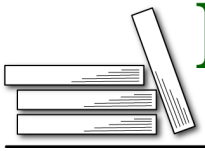
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Real-Time Phoenix

Build Highly Scalable Systems with Channels

Stephen Bussey

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Sean Dennis
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-719-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2020

Render Real-Time HTML with Channels

There are two major real-time features of our store. The first is to mark a shoe as released and to update all connected shoppers with the released shoe. We'll use HTML replacement for this feature by swapping out "coming soon" with our size selector. This approach makes it easy to ensure that a user interface looks the same before and after a real-time update occurs.

Adding the application's real-time features is usually less work than the other parts of writing the application due to Channel's abstractions. In this chapter, we'll write a small amount of code compared to the size of the project base that exists already. Real-time features are often added on top of other features, so it does make sense that you'll spend more time building the features and less time enhancing them to be real-time.

Our front end currently isn't connected to a Channel that could provide it with real-time updates. To start, we'll add a very simple Socket and Channel, and then connect our storefront to it. We'll leverage a Channel to send data from the server to a client. We don't need to add authentication because this is a public feature that anyone can see. There is no user-sensitive data in any of the Channels that we'll build in this chapter. Let's start by updating our Endpoint with a new Socket.

sneakers_23/lib/sneakers_23_web/endpoint.ex

```
socket "/product_socket", Sneakers23Web.ProductSocket,
  websocket: true,
  longpoll: false
```

You can replace the existing UserSocket definition with this one. UserSocket is one of the generated files that comes with Phoenix. You can optionally delete the channels/user_socket.ex file now. Let's define ProductSocket now.

sneakers_23/lib/sneakers_23_web/channels/product_socket.ex

```
defmodule Sneakers23Web.ProductSocket do
  use Phoenix.Socket

  ## Channels
  channel "product:*", Sneakers23Web.ProductChannel

  def connect(_params, socket, _connect_info) do
    {:ok, socket}
  end

  def id(_socket), do: nil
end
```

This is a very standard Socket defined without any authentication, because the feature is publicly accessible. Our ProductChannel will be equally simple for now.

sneakers_23/lib/sneakers_23_web/channels/product_channel.ex

```
defmodule Sneakers23Web.ProductChannel do
  use Phoenix.Channel

  alias Sneakers23Web.{Endpoint, ProductView}

  def join("product:" <> _sku, %{}, socket) do
    {:ok, socket}
  end
end
```

We're not doing anything exciting in this Channel yet. Let's change that by defining a broadcast function. This is a fairly interesting function because we're going to render our size selector HTML for a given product.

sneakers_23/lib/sneakers_23_web/channels/product_channel.ex

```
def notify_product_released(product = %{id: id}) do
  size_html = Phoenix.View.render_to_string(
    ProductView,
    "_sizes.html",
    product: product
  )

  Endpoint.broadcast!("product:#{id}", "released", %{
    size_html: size_html
  })
end
```

This technique allows us to render full pages or templates from anywhere in our Elixir application. This is a big advantage because all the template logic lives in Elixir, rather than being duplicated in JavaScript. We should write a test for this function.

sneakers_23/test/sneakers_23_web/channels/product_channel_test.exs

```
Line 1  defmodule Sneakers23Web.ProductChannelTest do
        use Sneakers23Web.ChannelCase, async: true
        alias Sneakers23Web.{Endpoint, ProductChannel}
        alias Sneakers23.Inventory.CompleteProduct
    5
        describe "notify_product_released/1" do
          test "the size selector for the product is broadcast" do
            {inventory, _data} = Test.Factory.InventoryFactory.complete_products()
            [_, product] = CompleteProduct.get_complete_products(inventory)
   10
            topic = "product:#{product.id}"
            Endpoint.subscribe(topic)
            ProductChannel.notify_product_released(product)

   15       assert_broadcast "released", %{size_html: html}
            assert html =~ "size-container__entry"
            Enum.each(product.items, fn item ->
              assert html =~ ~s(value="#{item.id}")
```

```
-       end)
20    end
-   end
- end
```

Our test subscribes to the notified topic, on line 12, so that any broadcasted messages will be received by the test process. This lets assert_broadcast check that the right message was broadcast. On line 18, our test ensures that each item of the product is accounted for in the HTML.

This function will be called whenever our item is released, which happens in the Inventory context. We'll use our Sneakers23Web module as our web context and will define a function that delegates to the ProductChannel. Elixir gives us a built-in way to do this.

sneakers_23/lib/sneakers_23_web.ex

```
defdelegate notify_product_released(product),
  to: Sneakers23Web.ProductChannel
```

The defdelegate macro[1] is incredibly useful for building a context module because it lets you separate implementation from exposure in a very quick and easy way. We now have to use this delegate function in our Inventory context. Without it, a product release event will not be broadcast to connected clients. Add the following test at the end of the existing describe block.

sneakers_23/test/sneakers_23/inventory_test.exs

```
test "the update is sent to the client", %{test: test_name} do
  {_, %{p1: p1}} = Test.Factory.InventoryFactory.complete_products()
  {:ok, pid} = Server.start_link(name: test_name, loader_mod: DatabaseLoader)
  Sneakers23Web.Endpoint.subscribe("product:#{p1.id}")

  Inventory.mark_product_released!(p1.id, pid: pid)
  assert_received %Phoenix.Socket.Broadcast{event: "released"}
end
```

You'll see this test fails when you run mix test. This is because the Inventory.mark_product_released!/2 function doesn't call notify_product_released/1. Let's fix that now.

sneakers_23/lib/sneakers_23/inventory.ex

```
def mark_product_released!(id), do: mark_product_released!(id, [])
def mark_product_released!(product_id, opts) do
  pid = Keyword.get(opts, :pid, __MODULE__)

  %{id: id} = Store.mark_product_released!(product_id)
  {:ok, inventory} = Server.mark_product_released!(pid, id)
  {:ok, product} = CompleteProduct.get_product_by_id(inventory, id)
  Sneakers23Web.notify_product_released(product)

  :ok
```

1. https://hexdocs.pm/elixir/Kernel.html#defdelegate/2

```
end
```

You can use default options in the function definition, like mark_product_released!(product_id, opts \\ []), instead of writing two separate function definitions. However, this book will often omit that type of definition.

All of the tests will now pass. This means that the back end is fully working. The Inventory context provides a function that marks the product as released in the database, changes it locally in the Inventory.Server process, then pushes the new state to any connected clients.

Now that our back end is configured, let's connect our front end by using the Phoenix Channel JavaScript client. Our strategy will be to grab the data-product-id attributes off of our HTML DOM elements and then connect to a Channel per matching product ID.

sneakers_23/assets/js/app.js

```
import css from "../css/app.css"
import { productSocket } from "./socket"
import dom from './dom'

const productIds = dom.getProductIds()

if (productIds.length > 0) {
  productSocket.connect()
  productIds.forEach((id) => setupProductChannel(productSocket, id))
}

function setupProductChannel(socket, productId) {
  const productChannel = socket.channel(`product:${productId}`)
  productChannel.join()
    .receive("error", () => {
      console.error("Channel join failed")
    })
}
```

This isn't a runnable example yet because we need to define our dom.js and socket.js files. However, the flow that we'll follow is complete. We'll soon add additional setup operations into setupProductChannel/1, which is why that function ends without closing.

sneakers_23/assets/js/socket.js

```
import { Socket } from "phoenix"

export const productSocket = new Socket("/product_socket")
```

This file simply makes the productSocket available for import. It's a good idea to keep the code separated with exported modules to help increase the focus of a particular file, even if there's no logic in the file now. It also gives us a place

to add more Socket-specific logic in the future, if needed. We still need to define our DOM operations.

sneakers_23/assets/js/dom.js

```
const dom = {}

function getProductIds() {
  const products = document.querySelectorAll('.product-listing')
  return Array.from(products).map((el) => el.dataset.productId)
}

dom.getProductIds = getProductIds

export default dom
```

This function will grab the matching .product-listing elements and return each productId attribute. At this point, everything is complete for our Socket to connect. Try it out by starting mix phx.server and visiting http://localhost:4000. You should see a Socket request in the "Network" tab as well as Channel join messages for product:1 and product:2. We're ready to wire up our product release message.

Start your server with iex -S mix phx.server so you can trigger the release message. Do so like this:

```
$ iex -S mix phx.server
iex(1)> {:ok, products} = Sneakers23.Inventory.get_complete_products()
iex(2)> List.last(products) |> Sneakers23Web.notify_product_released()
:ok
```

You can run this as many times as you want because it doesn't modify data. Try to watch the network message tab while you execute it. You should see the "released" message come through with an HTML payload. If you don't see it, make sure that you're inspecting the product_socket connection and not the live_reload connection.

Our front end needs to listen for this event in order to display the HTML.

sneakers_23/assets/js/app.js

```
function setupProductChannel(socket, productId) {
  const productChannel = socket.channel(`product:${productId}`)
  productChannel.join()
    .receive("error", () => {
      console.error("Channel join failed")
    })

  productChannel.on('released', ({ size_html }) => {
    dom.replaceProductComingSoon(productId, size_html)
  })
}
```

Our setup function is now adding a handler for the "released" event from the Channel. When the event is received, the DOM elements will be replaced with the new HTML. We'll add that function into our dom module, above the bottom export.

sneakers_23/assets/js/dom.js

```
function replaceProductComingSoon(productId, sizeHtml) {
  const name = `.product-soon-${productId}`
  const productSoonEls = document.querySelectorAll(name)

  productSoonEls.forEach((el) => {
    const fragment = document.createRange()
                        .createContextualFragment(sizeHtml)
    el.replaceWith(fragment)
  })
}

dom.replaceProductComingSoon = replaceProductComingSoon
```

We're not using jQuery or a similar library in this project. If we were, we could replace this HTML with something a bit simpler. This function lets the DOM turn HTML into the appropriate node types, and then swaps out the original element for the new node.

This is one of the more exciting parts of the demo! Our first real-time message is working end-to-end. Trigger notify_product_released/1 in the console when you have the page loaded. You will see the "coming soon" text instantly replaced by the shoe size selector, complete with the right colors. Type the following commands into your terminal.

```
$ mix ecto.reset && mix run -e "Sneakers23Mock.Seeds.seed!()"
$ iex -S mix phx.server
iex(1)> Sneakers23.Inventory.mark_product_released!(1)
iex(2)> Sneakers23.Inventory.mark_product_released!(2)
```

Take a moment to commit all of your current changes. The feature to release our product is fully implemented. This is a great time to make sure that you fully understand the code powering Sneakers23.Inventory.mark_product_released!/1 before moving on.

Next, you will implement another real-time feature in JavaScript, without HTML. This provides some variety in the way that you implement real-time features.