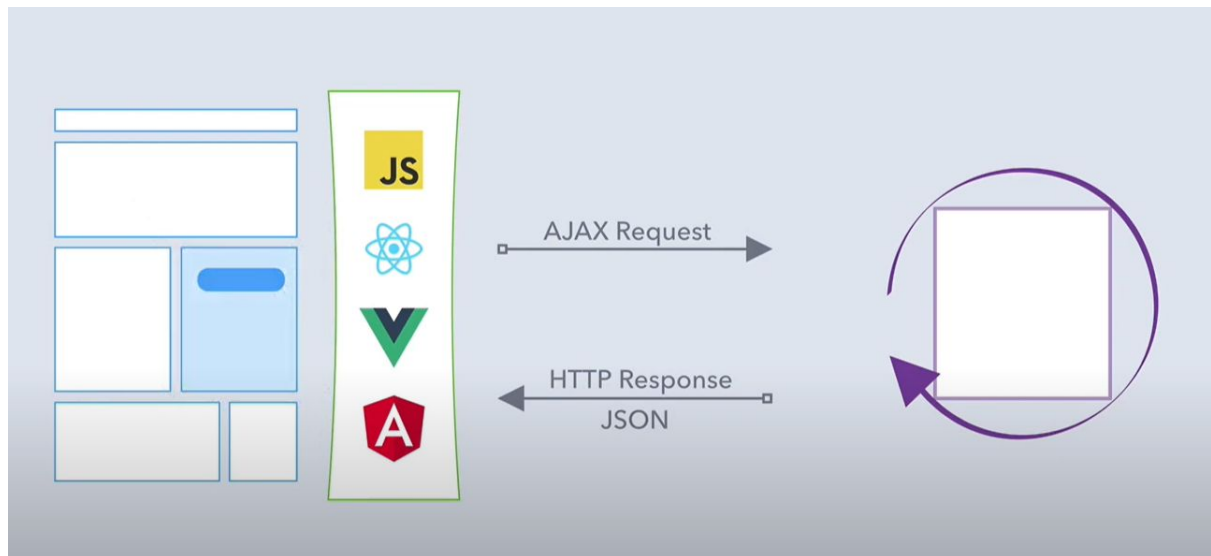
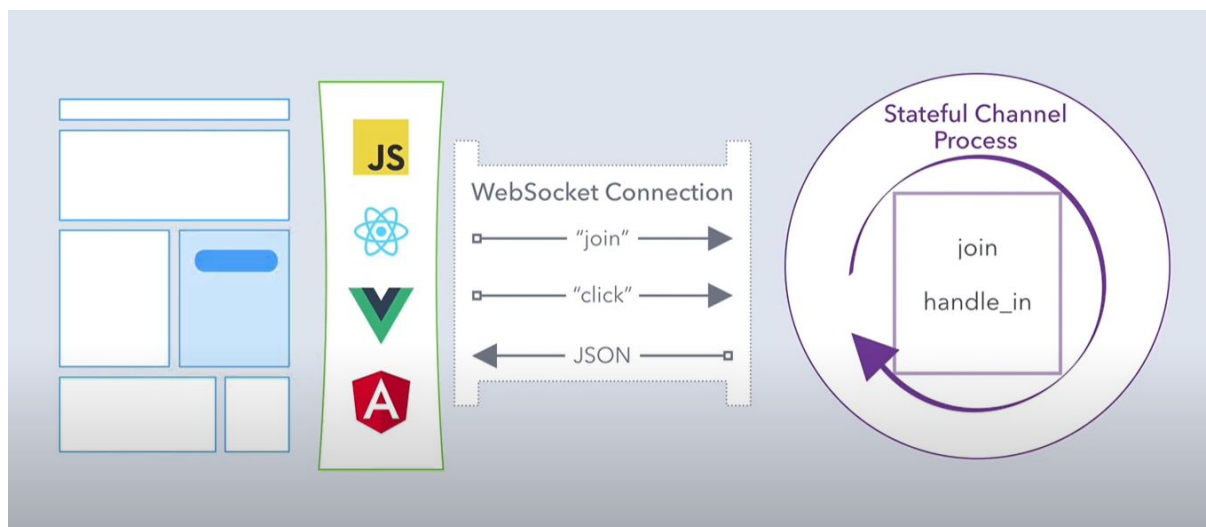


Cómo funciona Phoenix Live View?

Por lo general cuando queremos crear una aplicación web que sea capaz de modificar solo ciertas partes del dom usaremos el siguiente approach donde necesitaremos alguna tecnología del lado del cliente como JavaScript, Vue, Angular, React entre otras. estas enviarán AJAX Requests y el backend responderá con JSON.



Otro approach es hacer que el cliente se conecte con el server phoenix a través de un websocket, se genera un channel phoenix, js se subscribe a ese canal y empiezan a intercambiar mensajes, el server recibe eventos y cambia su estado acorde a estos, devolviendo así un JSON que otra vez es renderizado del lado del cliente para actualizar el dom.



Phoenix LiveView ofrece renderización del lado del server.

Otras tecnologías que implementan esto envían toda la página renderizada con cada request, en cambio Phoenix LiveView trackea cada cambio, separando además, partes estáticas de

partes dinámicas (aquellas que probablemente cambiarán debido a un evento).

PhoenixLiveView tiene un mínimo de JavaScript incluido para que nosotros desarrolladores solo nos ocupemos de escribir código Elixir.

Cada vez que levantemos una de estas vistas se iniciará un proceso el cual implementa tres callbacks:

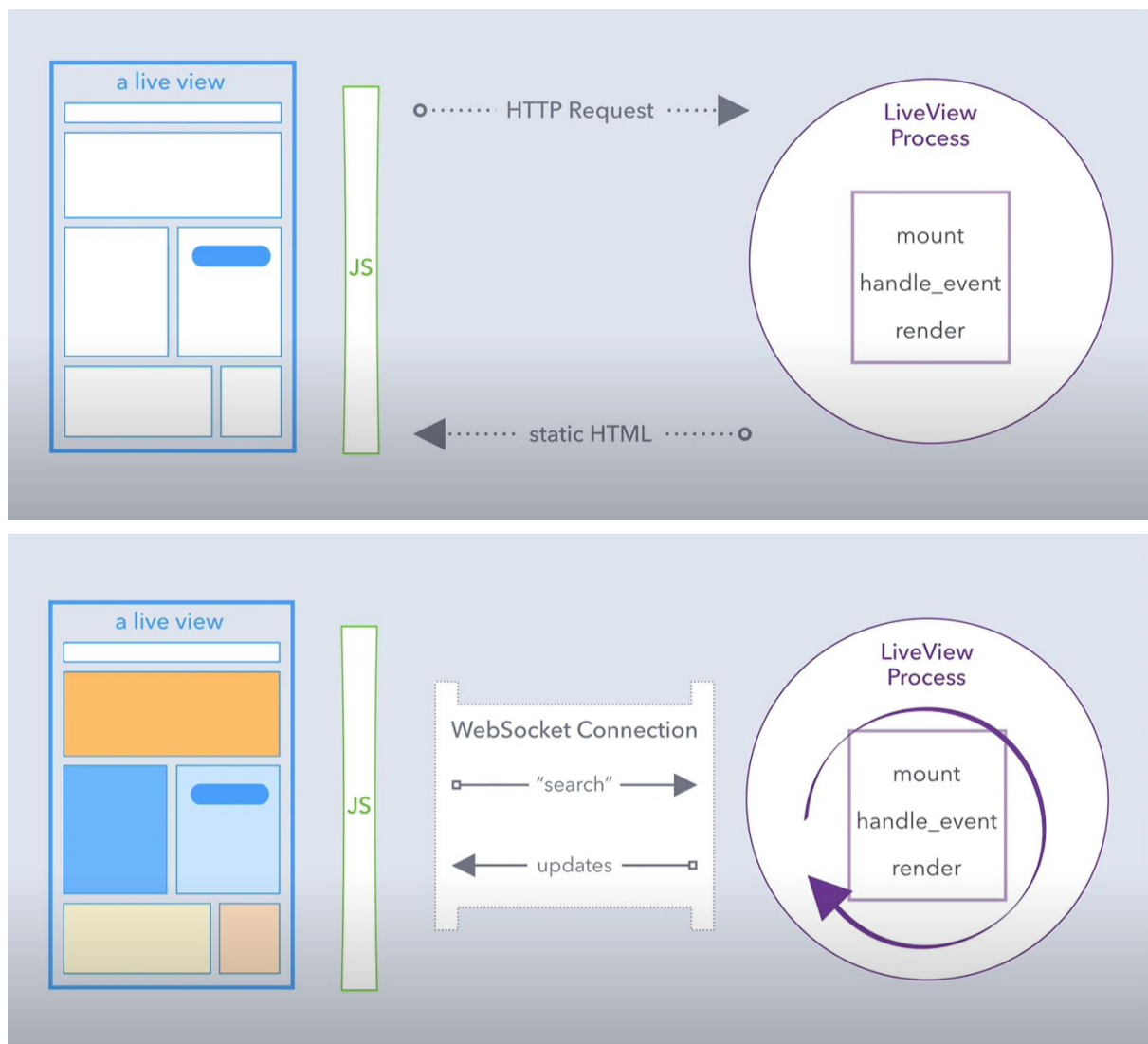
- mount: asigna el estado inicial del liveview process
- handle event: cambia el es estado del process
- render: renderiza una nueva vista para el nuevo estado

La primera vez recibirá un HTTP Request al cual responderá con un template de HTML estático. Luego se establecerá la conexión a través de un websocket y se levantará un proceso

LiveView,

se invocará a la callback mount asignando el estado inicial y se empiezan a recibir eventos a través del websocket.

Phoenix LiveView nos garantiza el intercambio de la información justa y necesaria haciendo nuestras aplicaciones super performantes



Personalmente preferí no basar mi proyecto integrador en una app web porque para implementar apps en Phoenix necesitamos escribir código del lado del cliente (y yo no tengo idea de como hacer eso), investigando un poco mas me topé con este feature Phoenix LiveView el cual sigue en desarrollo y hace poco salió su última versión.

Entendiendo un poco de cómo se manejan los procesos en Elixir pude comprender, a grandes

rasgos, cómo se construían estas vistas.

Finalmente me di cuenta de que podemos construir nuestra primera app con LiveView en simples pasos y en pocos minutos

Counter Live

Con el siguiente comando generaremos la estructura de nuestro proyecto

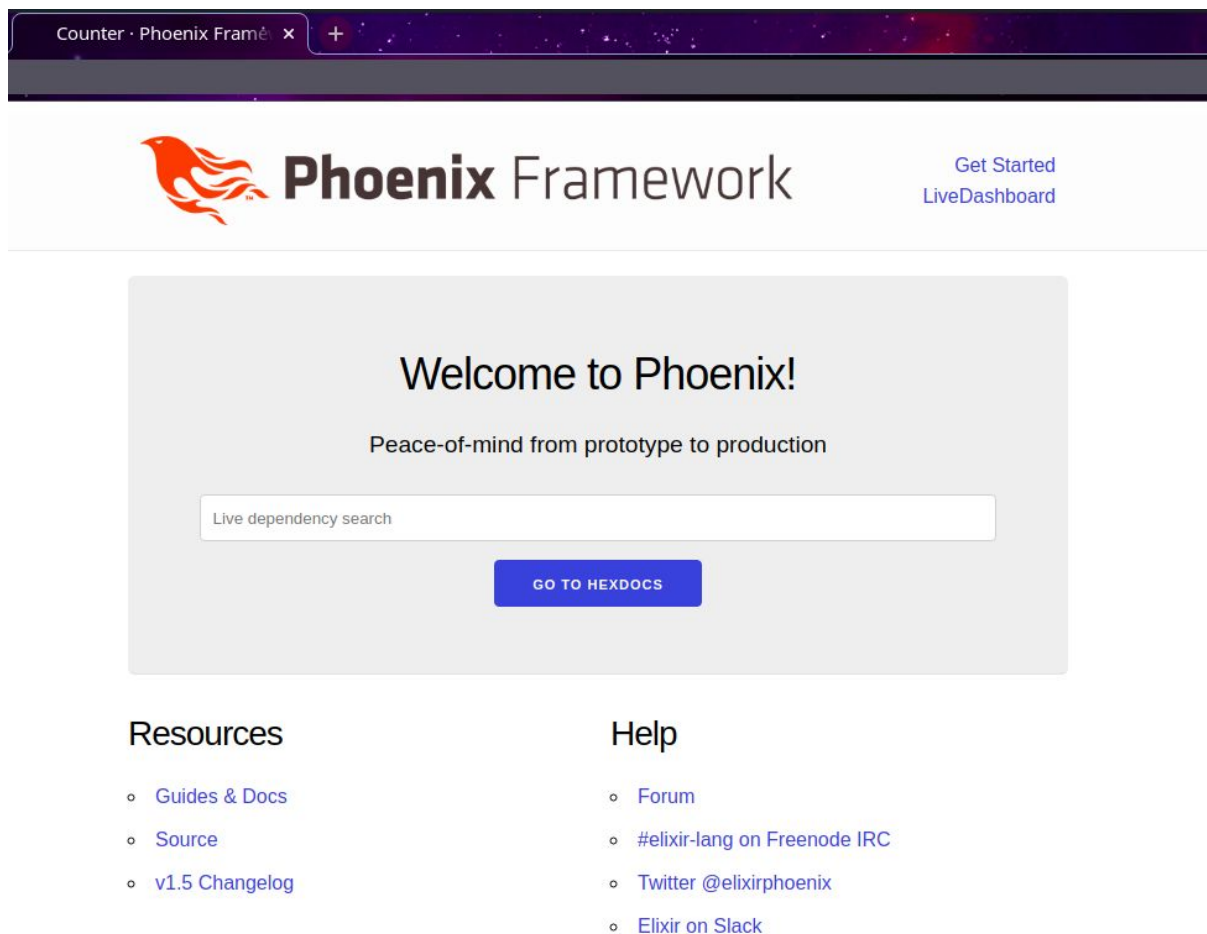
```
mix phx.new counter --live
```

```
cd counter/
```

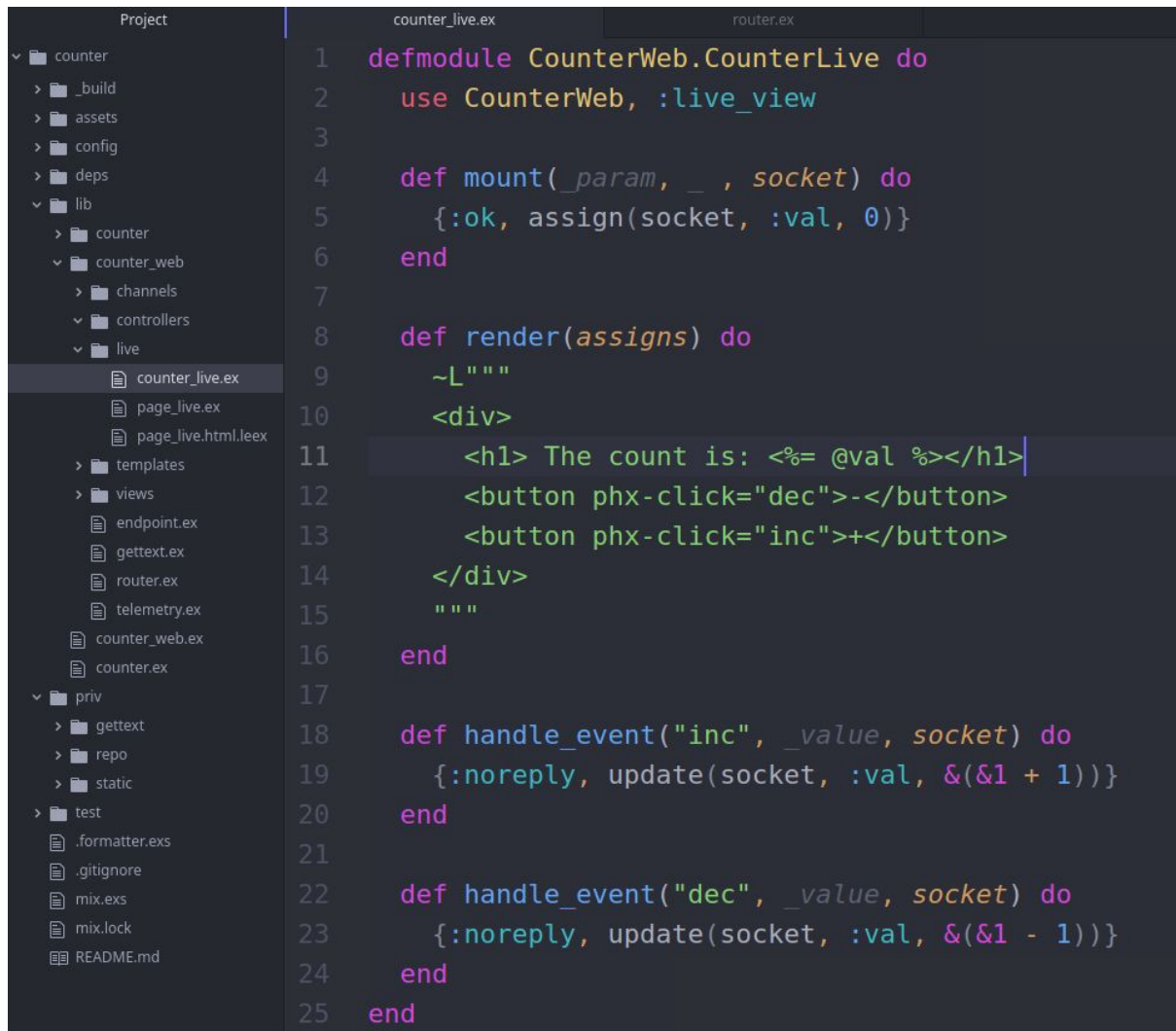
una vez en el directorio de nuestro proyecto levantamos nuestra app usando

```
mix phx.server una vez
```

Abrimos nuestro browser “localhost:4000” y nos encontraremos con la vista predeterminada del framework.



Ahora veamos el código::



```
1 defmodule CounterWeb.CounterLive do
2   use CounterWeb, :live_view
3
4   def mount(_param, _ , socket) do
5     {:ok, assign(socket, :val, 0)}
6   end
7
8   def render(assigns) do
9     ~L"""
10     <div>
11       <h1> The count is: <%= @val %></h1>
12       <button phx-click="dec">-</button>
13       <button phx-click="inc">+</button>
14     </div>
15     """
16   end
17
18   def handle_event("inc", _value, socket) do
19     {:noreply, update(socket, :val, &(&1 + 1))}
20   end
21
22   def handle_event("dec", _value, socket) do
23     {:noreply, update(socket, :val, &(&1 - 1))}
24   end
25 end
```

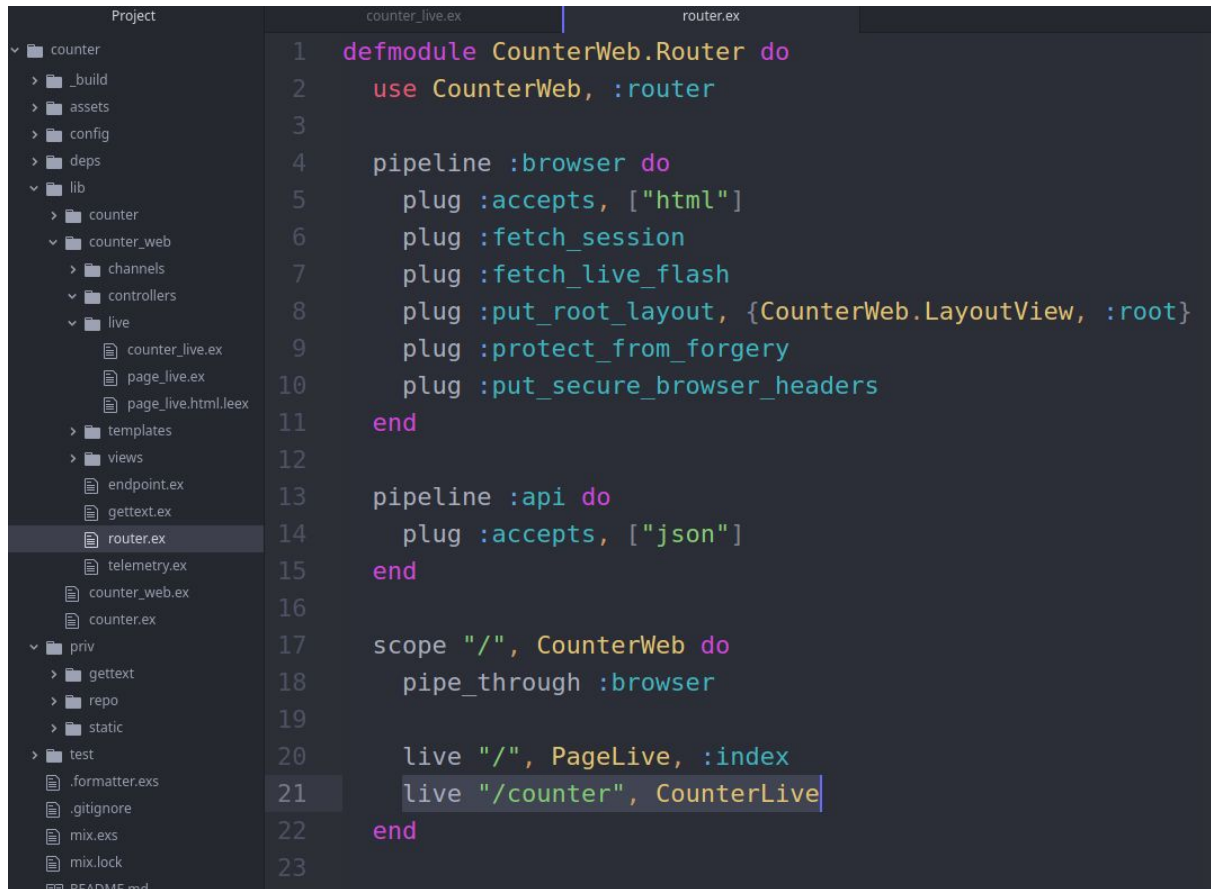
De esta manera se ve la implementación de nuestro contador:

Mount va a asignar el estado inicial al proceso llamando a la función assign la cual genera la estructura donde se guarda toda la información necesaria que transportamos a través del websocket.

Render va a trabajar con ese estado reemplazando valores en el template que le asignemos para así poder renderizar nuestras vistas, como podemos observar vamos a tener un display con el valor actual de nuestro contador seguido de dos botones que al ser clickeados enviaran un evento en forma de callback, estos serán atendidos en sus respectivos handle_event

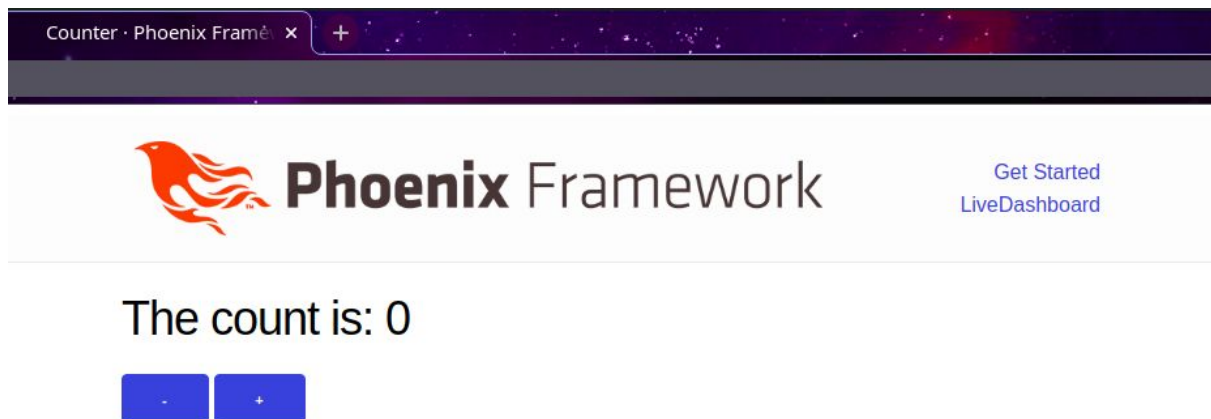
Handle_event procesa mensajes recibidos como vimos en el módulo de GenServer , en este caso matcheando con el nombre del evento generado, en este caso tenemos un botón de suma 1 al valor actual y otro que resta 1, este cambio en el estado del proceso se aplica a través de la función update.

Bien y ahora por último vamos a agregar una ruta por la cual vamos a acceder a nuestra vista, está se indica con la macro `live`, un string con el path y el nombre del proceso donde está levantada esa vista.

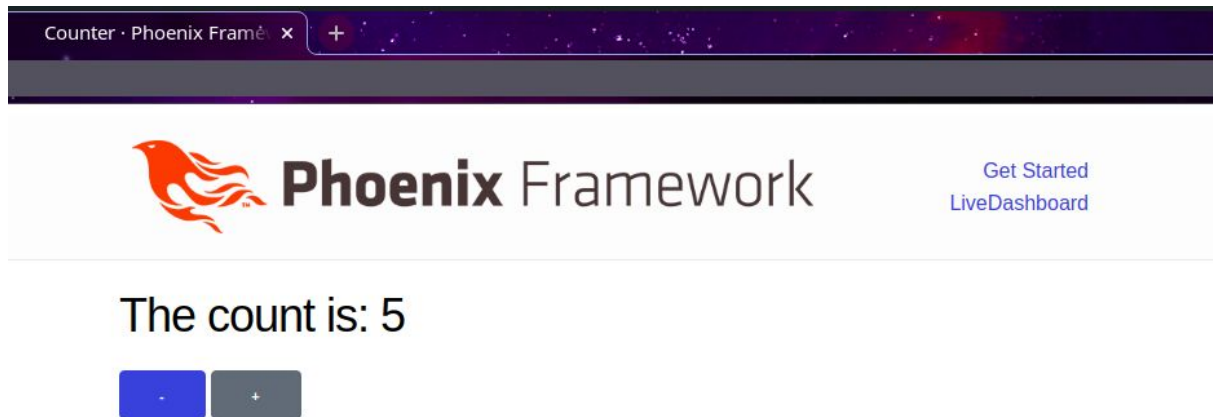


```
1 defmodule CounterWeb.Router do
2   use CounterWeb, :router
3
4   pipeline :browser do
5     plug :accepts, ["html"]
6     plug :fetch_session
7     plug :fetch_live_flash
8     plug :put_root_layout, {CounterWeb.LayoutView, :root}
9     plug :protect_from_forgery
10    plug :put_secure_browser_headers
11  end
12
13  pipeline :api do
14    plug :accepts, ["json"]
15  end
16
17  scope "/", CounterWeb do
18    pipe_through :browser
19
20    live "/", PageLive, :index
21    live "/counter", CounterLive
22  end
23
```

Volvemos al browser, esta vez “localhost:4000/counter”



Y después de haber clickeado 5 veces el botón de incremento...



Así de sencillo y sin tenerle miedo al uso de alguna tecnología extra pude empezar a jugar con este framework y como lo prometieron solamente tuve que escribir código elixir.