Extracted from:

Craft GraphQL APIs in Elixir with Absinthe

Flexible, Robust Services for Queries, Mutations, and Subscriptions

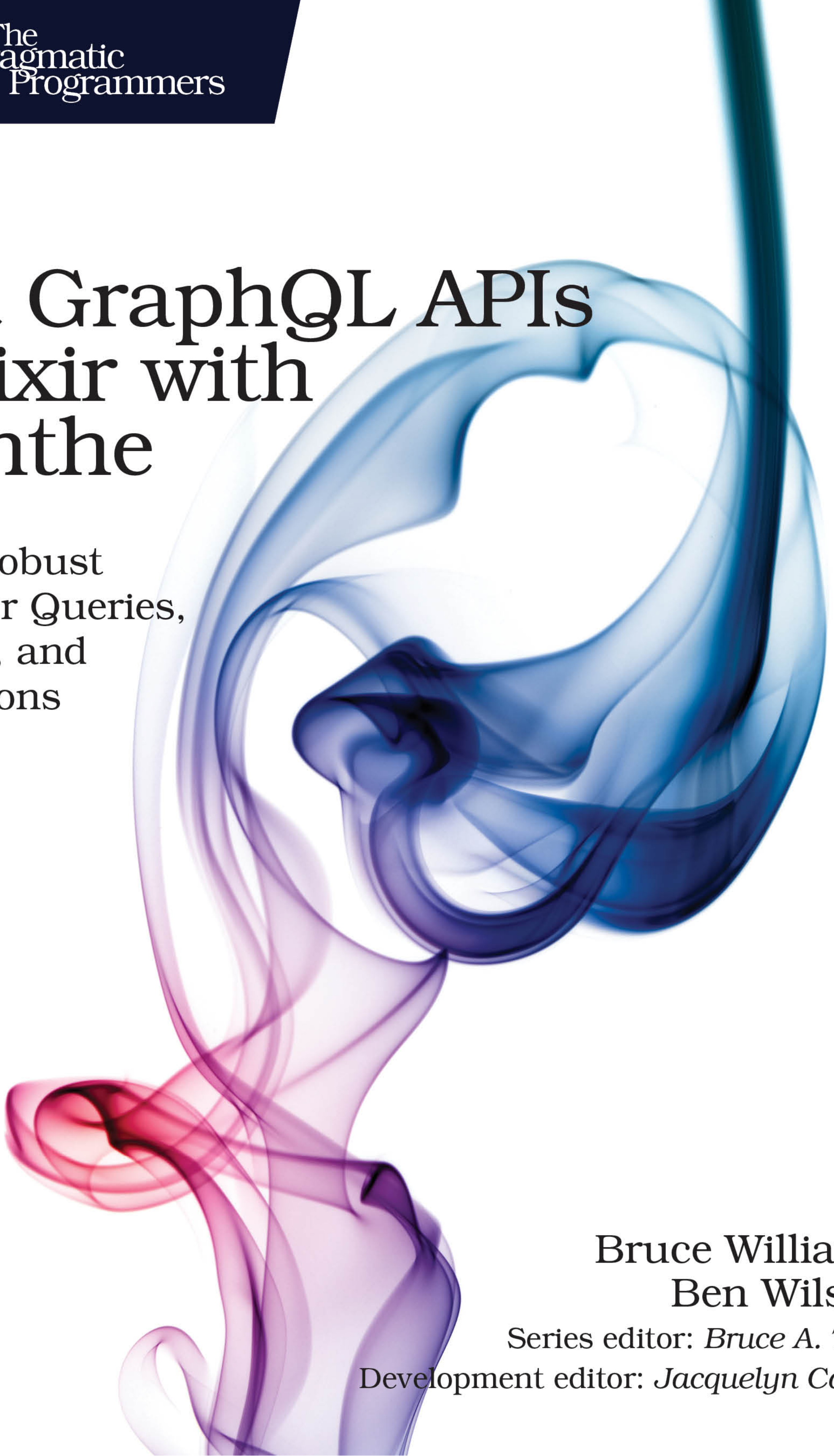
The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Craft GraphQL APIs in Elixir with Absinthe

Flexible, Robust Services for Queries, Mutations, and Subscriptions



Bruce Williams
Ben Wilson

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

Craft GraphQL APIs in Elixir with Absinthe

Flexible, Robust Services for Queries, Mutations, and Subscriptions

Bruce Williams
Ben Wilson

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Series Editor: Bruce A. Tate
Copy Editor: Nicole Abramowitz
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: 978-1-68050-255-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—April 2020

Submitting Subscriptions

Now that we can create orders, we're at the perfect spot to introduce the first basic subscription that will support pushing these orders as they're created out to subscribed clients. You'll first need to define a subscription field in your schema, and then you'll also need a way to actually trigger this subscription when the :place_order mutation runs.

06-chp.subscriptions/4-publish/lib/plate_slate_web/schema.ex

```
subscription do
  field :new_order, :order do

    config fn _args, _info ->
      {:ok, topic: "*"}
    end
  end
end
```

For the most part, this is a pretty ordinary-looking field. We've got another top-level object, subscription, to house our subscription fields, and then the :new_order, which will return the :order object we're already familiar with. The fact that it returns a regular :order object is crucial, because this means that all the work we have done to support the output of the mutation can be reused immediately for real-time data.

What's new, however, is the config macro, and it's one of a couple macros that are specific to setting up subscriptions. The job of the config macro is to help us determine which clients who have asked for a given subscription field should receive data by configuring a *topic*. We'll talk more later about constructing topics, but the main thing to know is that topics are scoped to the field they're on, and they have to be a string. We're just going to use "*" to indicate that we care about all orders (but there's nothing special about "*" itself).

Set Up and Return Error

The config function can also return {:error, reason}, which prevents the subscription from being created.

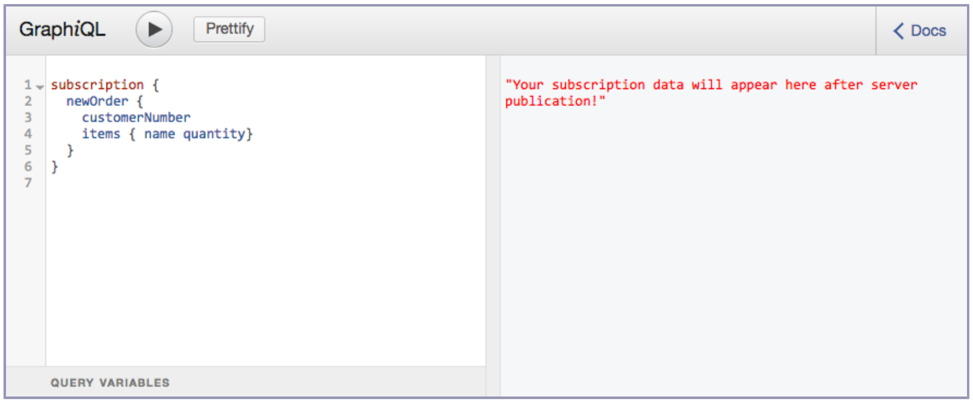
Let's see if we can subscribe with GraphiQL. First, let's make sure it's running again, but this time, inside an IEx session (you'll see why shortly):

```
$ iex -S mix phx.server
[info] Running PlateSlateWeb.Endpoint with Cowboy using http://0.0.0.0:4000
```

Then, browse to http://localhost:4000/graphiql and enter the following in the left-side panel (you can close the "Query Variables" panel if you have it open):

```
subscription {
  newOrder {
    customerNumber
    items { name quantity}
  }
}
```

When you hit “play,” instead of getting a result, you’ll get a message saying, "Your subscription data will appear here after server publication!":



```
subscription {
  newOrder {
    customerNumber
    items { name quantity}
  }
}
```

```
"Your subscription data will appear here after server
publication!"
```

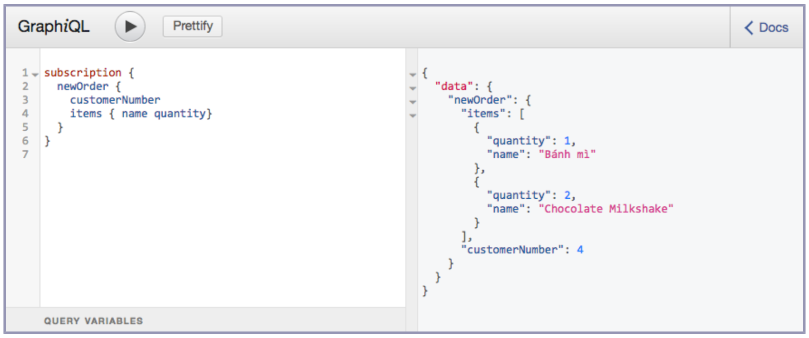
What’s happening here is that although the server has accepted the subscription document, the server is waiting on some kind of event that will trigger execution of the document and distribution of the result. Specifically, it’s waiting for an event that targets the field of our subscription newOrder and the topic associated with this specific document "*".

The most direct way to make this trigger happen is with the Absinthe.Subscription.publish/3 function, which gives us manual control of the publishing mechanism. If you go into the IEx session in your console, you can trigger the subscription you just created in GraphiQL by running:

```
iex> order = PlateSlate.Ordering.Order |> PlateSlate.Repo.all |> List.last
«%PlateSlate.Ordering.Order{} displayed»
iex> Absinthe.Subscription.publish(
  PlateSlateWeb.Endpoint,
  order,
  new_order: "*"
)
:ok
```

If you look back to your GraphiQL page, you should see a result as shown in the figure on page 7.

The arguments to the publish/3 function are the module you’re using as the pubsub, the value that you’re broadcasting, and the field: topic pairs at which



```
subscription {
  newOrder {
    customerNumber
    items { name quantity}
  }
}
```

```
{
  "data": {
    "newOrder": {
      "items": [
        {
          "quantity": 1,
          "name": "Bánh mì"
        },
        {
          "quantity": 2,
          "name": "Chocolate Milkshake"
        }
      ],
      "customerNumber": 4
    }
  }
}
```

to broadcast the value. Concretely then, the function call you typed in IEx says to broadcast the last %Order{} struct to all clients subscribed to the :new_order field via the "*" topic.

You may have noticed when you set up the subscription field that you didn't specify a resolver, and this is why. Unlike a root query or root mutation resolver, which generally starts with no root value and has to start from scratch, the root value of a subscription document is the value that is passed to publish/3. You can see this for yourself if you add just an inspect resolver to the subscription field:

06-chp.subscriptions/4-publish/lib/plate_slate_web/schema.ex

```
subscription do
  field :new_order, :order do

    config fn _args, _info ->
      {:ok, topic: "*"}
    end

➤   resolve fn root, _, _ ->
➤     IO.inspect(root)
➤     {:ok, root}
➤   end
  end
end
```

If you re-run the Absinthe.Subscription.publish/3 call in your IEx session, you will see printed into console the value you are broadcasting, nested under the :new_order key:

```
%Ordering.Order{
  «Contents»
}
```

With this Absinthe.Subscription.publish/3 function at our disposal, it's clear then that one possibility for making our live interface is to put it inside the :place_order mutation resolver so that instead of triggering subscriptions from IEx, we'll trigger subscriptions every time a new order is placed.

06-chp.subscriptions/4-publish/lib/plate_slate_web/resolvers/ordering.ex

```
def place_order(_, %{input: place_order_input}, _) do
  case Ordering.create_order(place_order_input) do
    {:ok, order} ->
➤      Absinthe.Subscription.publish(PlateSlateWeb.Endpoint, order,
➤        new_order: "*"
➤      )
      {:ok, %{order: order}}
    {:error, changeset} ->
      {:ok, %{errors: transform_errors(changeset)}}
  end
end
```

You could have some fun with this:

- In one window, open GraphiQL and enter the subscription document.
- In another window, also go to GraphiQL and enter the mutation document.
- Press "play" in the mutation window.
- Watch real-time events show up in the subscription window!

Play around with changing what parts of the subscription document you ask for, and play around with the values you put in the mutation document to get a feel for how the two documents relate to one another.

Testing Subscriptions

Testing your API is important, and subscriptions are no exception. We've been using helpers from the PlateSlate.ConnCase module in our test to ease building HTTP-based integration tests; you'll need a similar PlateSlate.SubscriptionCase module for managing the subscription integration tests via channels. While the ConnCase module gets generated by Phoenix when we first create the project, the SubscriptionCase module we'll need to make ourselves.

06-chp.subscriptions/4-publish/test/support/subscription_case.ex

```
defmodule PlateSlateWeb.SubscriptionCase do
  @moduledoc """
  This module defines the test case to be used by
  subscription tests
  """

  use ExUnit.CaseTemplate

  using do
    quote do
      # Import conveniences for testing with channels
      use PlateSlateWeb.ChannelCase
      use Absinthe.Phoenix.SubscriptionTest,
        schema: PlateSlateWeb.Schema
      setup do
        PlateSlate.Seeds.run()

        {:ok, socket} =
            Phoenix.ChannelTest.connect(PlateSlateWeb.UserSocket, %{})
        {:ok, socket} =
            Absinthe.Phoenix.SubscriptionTest.join_absinthe(socket)

        {:ok, socket: socket}
      end

      import unquote(__MODULE__), only: [menu_item: 1]
    end
  end

  # handy function for grabbing a fixture
  def menu_item(name) do
    PlateSlate.Repo.get_by!(PlateSlate.Menu.Item, name: name)
  end
end
```

This module sets up the socket we'll use in each of our test cases, and it also gives us a convenient function for getting menu items.

As far as the test case itself goes, much of the setup here is exactly the same as any other Phoenix channel test. Absinthe.Phoenix provides some helpers to instantiate a socket process with the configuration you added to the UserSocket.

06-chp.subscriptions/4-publish/test/plate_slate_web/schema/subscription/new_order_test.exs

```
defmodule PlateSlateWeb.Schema.Subscription.NewOrderTest do
  use PlateSlateWeb.SubscriptionCase

  @subscription """
  subscription {
    newOrder {
      customerNumber
    }
  }
  """
  @mutation """
  mutation ($input: PlaceOrderInput!) {
    placeOrder(input: $input) { order { id } }
  }
  """
  test "new orders can be subscribed to", %{socket: socket} do
    # setup a subscription
    ref = push_doc socket, @subscription
    assert_reply ref, :ok, %{subscriptionId: subscription_id}

    # run a mutation to trigger the subscription
    order_input = %{"customerNumber" => 24,
      "items" => [%{"quantity" => 2, "menuItemId" => menu_item("Reuben").id}]
    }
    ref = push_doc socket, @mutation, variables: %{"input" => order_input}
    assert_reply ref, :ok, reply
    assert %{data: %{"placeOrder" => %{"order" => %{"id" => _}}}} = reply

    # check to see if we got subscription data
    expected = %{
      result: %{data: %{"newOrder" => %{"customerNumber" => 24}}},
      subscriptionId: subscription_id
    }
    assert_push "subscription:data", push
    assert expected == push
  end
end
```

If channels are pretty new to you, that's okay. The essential thing to keep in mind is that it's a lot like testing a GenServer. You've got the test process itself, which acts like the client, and you've got the socket process, which operates just as it does when connected to by an external client. You can push an event and params to the socket and then listen for a specific reply to that event, much like a GenServer call. Each push is asynchronous, so it's important to make sure to wait for a reply after each push. The socket can also send messages directly to the test process, which is what will happen when we trigger an event.

Testing a socket, then, is just a matter of sending it the data we need to configure our subscription, triggering a mutation, and then waiting for subscription data to get pushed to the test process.

So the first thing we do is push a "doc" event to the socket along with the parameters specifying our subscription document, and then we assert for a reply from the socket that returns a subscriptionId. This subscriptionId is important because a single socket can support many different subscriptions, and the subscriptionId is used to keep track of what data push belongs to what subscription.

The next thing we do is run a mutation to place an order. This operation is actually pushed over the socket as well; sockets support all the different operation types. While an explicit Absinthe.run would also work, it would require that we explicitly pass in the pubsub configuration, whereas that config is picked up automatically if the document is pushed through the socket.

Finally, all we have to do is assert that the test process gets a message containing the expected subscription data!

Let's go ahead and run our test to make sure everything is working as expected:

```
$ mix test test/plate_slate_web/schema/subscription/new_order_test.exs
«Debugging output»
.

Finished in 0.1 seconds
1 test, 0 failures
```

There we go! Not only did we get subscriptions working in GraphiQL, we were able to treat it like any other part of our API and write a proper integration test. Now that we have subscriptions working using a manual method, let's look at a mechanism that we can use to publish changes automatically as they occur.

Subscription Triggers

In the previous section, we only used a single hard-coded topic value, but when we start thinking about tracking the life cycle of a particular entity, we need to pay a lot more attention to how we're setting up our subscriptions and how we're triggering them. The challenge isn't just keeping track of how the topics are constructed; it can also be hard to make sense of where in your code base publish/3 calls may be happening. We're going to explore an alternative approach to trigger mutations as we expand on the order-tracking capabilities of the PlateSlate system.

Everything that has a beginning has an end, and for the hungry customer, orders are fortunately no exception. We need to complete the life cycle of an order by providing two mutations: one to indicate that it's ready, and one to indicate that it was picked up.

Fortunately, most of what we need to do this in our context and schema already exists, so we can just jump directly to building out the relevant mutation fields in the GraphQL schema and filling out each resolver.

06-chp.subscriptions/5-trigger/lib/plate_slate_web/schema.ex

```
mutation do

  field :ready_order, :order_result do
    arg :id, non_null(:id)
    resolve &Resolvers.Ordering.ready_order/3
  end
  field :complete_order, :order_result do
    arg :id, non_null(:id)
    resolve &Resolvers.Ordering.complete_order/3
  end

  # «Other fields»
end
```

Our :ready_order and :complete_order fields use new resolver functions from PlateSlateWeb.Resolvers.Ordering; let's add those:

06-chp.subscriptions/5-trigger/lib/plate_slate_web/resolvers/ordering.ex
```
def ready_order(_, %{id: id}, _) do
  order = Ordering.get_order!(id)
  with {:ok, order} <- Ordering.update_order(order, %{state: "ready"}) do
    {:ok, %{order: order}}
  else
    {:error, changeset} ->
      {:ok, %{errors: transform_errors(changeset)}}
  end
end

def complete_order(_, %{id: id}, _) do
  order = Ordering.get_order!(id)

  with {:ok, order} <- Ordering.update_order(order, %{state: "complete"}) do
    {:ok, %{order: order}}
  else
    {:error, changeset} ->
      {:ok, %{errors: transform_errors(changeset)}}
  end
end
```

So far, so good. This may start to feel pretty second nature at this point. If you are concerned that the changeset error handling here is seeming kind of redundant, hold on tight—that is covered in the very next chapter on middleware.

Subscribing to these events is just a little bit different than before, because now we're trying to handle events for specific orders based on ID. When the client is notified about new orders via a new_order subscription, we then want to give them the ability to subscribe to future updates for each of those subscriptions specifically.

We want to support a GraphQL document that looks like:

```
subscription {
  updateOrder(id: "13") {
    customerNumber
    state
  }
}
```

Notably, we want to use this one subscription field to get updates triggered by both the :ready_order and :complete_order mutation fields. While it's important to represent the mutations as different fields, it's often the case that you just need a single subscription that lets you get all the state changes for a particular entity that you want to watch.

06-chp.subscriptions/5-trigger/lib/plate_slate_web/schema.ex
```
subscription do
  field :update_order, :order do
    arg :id, non_null(:id)

    config fn args, _info ->
      {:ok, topic: args.id}
    end
  end

  # «Other fields»
end
```

The main difference is that we're now doing something more dynamic in our config function. Here we're using the arguments provided to the field to generate a topic that is specific to the ID of the order we care about.

Based on your previous experience with the Absinthe.Subscription.publish/3 function, you might be able to figure out the function call you could put in each mutation resolver to trigger this subscription field:

```
Absinthe.Subscription.publish(
  PlateSlateWeb.Endpoint, order,
  update_order: order.id
)
```

However, while we could use the publish/3 function here, we're going to explore a slightly different option. The issue with our approach thus far is that although our schema contains the :place_order mutation and also the :new_order subscription fields, there isn't any indicator in the schema that these two fields are connected in any way. Moreover, for subscription fields that are triggered by several different mutations, the topic logic is similarly distributed in a way that can make it difficult to keep track of.

This pattern of connecting mutation and subscription fields to one another is so common that Absinthe considers it a first-class concept and supports setting it as a trigger on subscription fields, avoiding the need to scatter publish/3 calls throughout your code base. Let's look at how we can use the trigger macro to connect the new subscription field to each mutation without touching our resolvers:

06-chp.subscriptions/5-trigger/lib/plate_slate_web/schema.ex
```
subscription do
  field :update_order, :order do
    arg :id, non_null(:id)

    config fn args, _info ->
      {:ok, topic: args.id}
    end
    trigger [:ready_order, :complete_order], topic: fn
      %{order: order} -> [order.id]
      _ -> []
    end

    resolve fn %{order: order}, _ , _ ->
      {:ok, order}
    end
  end

  # «Other fields»
end
```