Extracted from:

Build a Weather Station with Elixir and Nerves

Visualize Your Sensor Data with Phoenix and Grafana

The Pragmatic Bookshelf

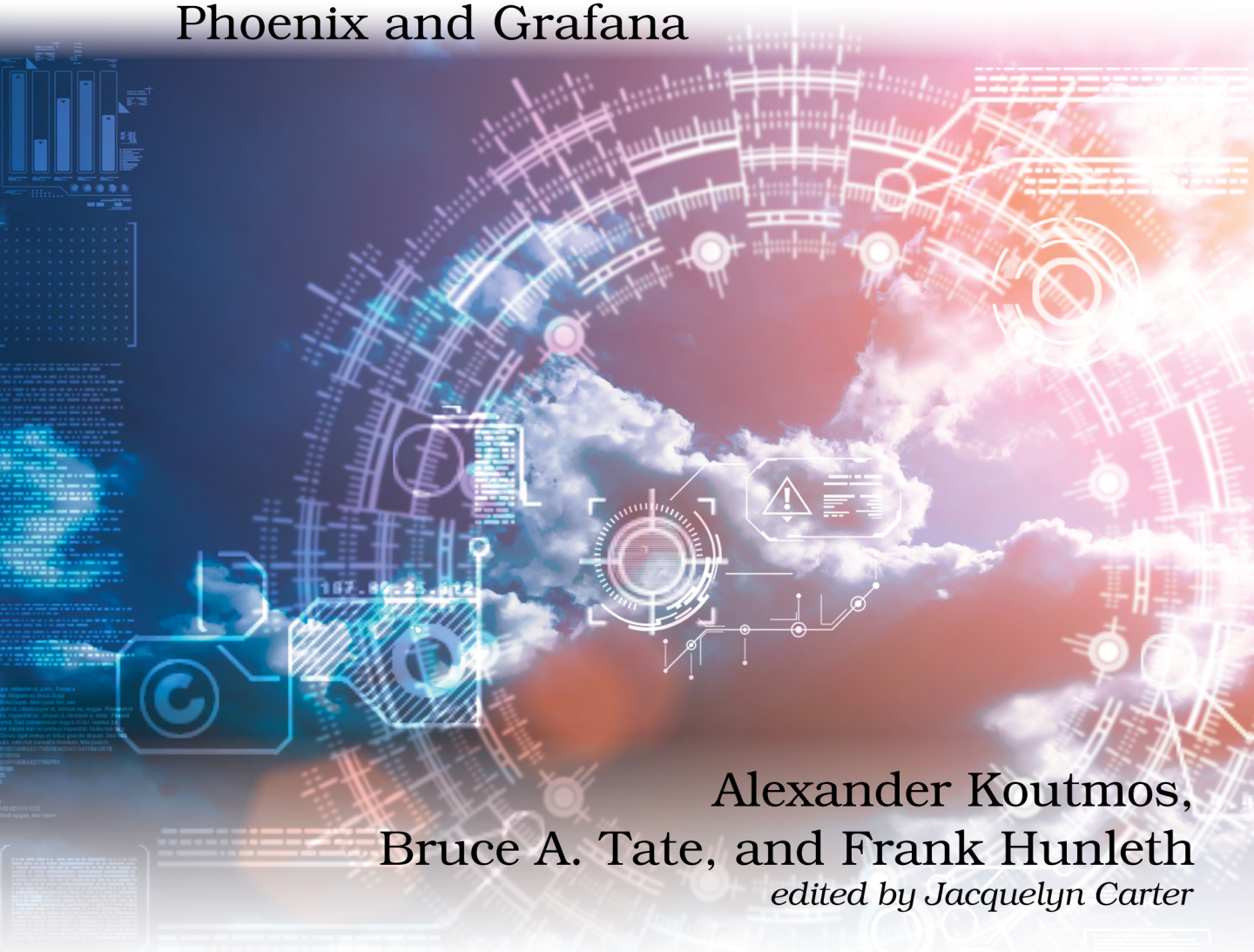
Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Build a Weather Station with Elixir and Nerves

Visualize Your Sensor Data with Phoenix and Grafana



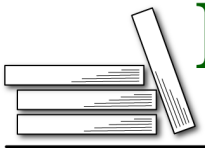
Alexander Koutmos,
Bruce A. Tate, and Frank Hunleth
edited by Jacquelyn Carter

Build a Weather Station with Elixir and Nerves

Visualize Your Sensor Data with Phoenix and Grafana

Alexander Koutmos
Bruce A. Tate
Frank Hunleth

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: L. Sakhi MacMillan
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

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

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

ISBN-13: 978-1-68050-902-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2022

Managing the Life Cycle

The application.ex file is often the first module called by Elixir within a new application. It implements a supervisor and is a convenient point for attaching initial startup features. For the purposes of this project, we also need to start the sensors and server publisher services (we'll tackle the publisher in the next chapter). Most of the configuration will go in the application.ex file as arguments to each of the new GenServers we start.

Setting up the Supervision Tree

Open up lib/sensor_hub/application.ex in the sensor_hub subproject, and let's fill out the small functions that will start the pieces of the firmware project. Let's begin with the children/1 function (specifically the version that pattern matches on _target):

```
defmodule SensorHub.Application do
  ...

  def children(_target) do
    # The sensors will fail on the host so let's
    # only start them on the target devices.
    [
      {SGP30, []},
      {BMP280, [i2c_address: 0x77, name: BMP280]},
      {VEML6030, %{}}
    ]
  end

  ...
end
```

This code should look slightly familiar. If you recall from our IEx sessions, when we started the sensor GenServers manually, we had to pass certain options to their start_link/1 functions for them to work properly. Instead of doing that ourselves manually on application start, we'll let our application supervisor take care of that now.

Let's also update our start/2 callback and ensure that it looks like so:

```
defmodule SensorHub.Application do
  ...

  @impl true
  def start(_type, _args) do
    # See https://hexdocs.pm/elixir/Supervisor.html
    # for other strategies and supported options
    opts = [strategy: :one_for_one, name: SensorHub.Supervisor]

    children = children(target())
```

```
    Supervisor.start_link(children, opts)
  end

  ...
end
```

As we can see here, our supervisor is provided the list of child processes, depending on what target we're running on. If you look at the Nerves-generated code, the version of children/1 that matches on :host has no child processes,

while our Raspberry Pi target contains all of our sensors (this value is provided by the target/0 function). This means that any time we run this project on our workstation, we won't be starting up any of our sensor GenServers, which makes sense given that the sensors are only supposed to run when the project is running on our Raspberry Pi.

With our application life-cycle code set up in our supervisor, it's time to test it all out and make sure that things start up and behave as expected.

Trying It Out

With our code up-to-date, it's time to create an up-to-date firmware via mix firmware, upload the firmware with mix upload hub.local, and then connect to the device with ssh hub.local. After we connect to the device, we can run the following commands to introspect the device and ensure that our sensor GenServers are up and running:

```
iex(1)> Supervisor.which_children(SensorHub.Supervisor)
[
  {VEML6030, #PID<0.1287.0>, :worker, [VEML6030]},
  {BMP280, #PID<0.1286.0>, :worker, [BMP280]},
  {SGP30, #PID<0.1285.0>, :worker, [SGP30]}
]

iex(2)> alias SensorHub.Sensor
SensorHub.Sensor

iex(3)> BMP280 |> Sensor.new() |> Sensor.measure()
%{
  altitude_m: 77.29732484024902,
  pressure_pa: 99087.08904119114,
  temperature_c: 25.63860611162145
}
```

By running the which_children/1 call in IEx, we're able to see what child processes are under the provided supervisor. As you can see, our Nerves application started up all of the GenServers that were specified in the application.ex file, and you were even able to interact with the BMP280 sensor to get sensor data.

Your Turn

In this chapter, we took what we learned from the previous sections and put it all together to create a Nerves application that starts up by itself, initializes all of the sensor hardware, and refreshes sensor measurements automatically.

What You Built

You started the chapter by creating the `veml6030` subproject and creating the stateless components to work with your light sensor. These stateless components were derived from the VEML6030 spec sheet and were needed to configure and communicate with the sensor. Once you had these things in place, you added a stateful element to the mix—namely the `VEML6030` GenServer module. This GenServer would regularly poll the sensor and store the results in its state. You could then read from this state at any point to get an up-to-date reading on the ambient light in the room.

After creating the `veml6030` sensor subproject, you were able to lean on the Elixir and Nerves ecosystem to pull down libraries for working with the additional weather station sensors. You then configured your application supervision tree to start up all of your sensors on device `init`, and added some glue code to make it easy to fetch data from all of the sensors.

Why It Matters

This chapter walked through exactly how to structure your Nerves applications so that they are bulletproof and production ready. While SSHing into devices and configuring them ad hoc is acceptable for development and experimentation, we need to leverage the OTP available to us to create a reliable and fault-tolerant IoT experience. By using GenServers and Supervisors, we were able to accomplish just that.

What's Next

Now that your Nerves application is almost complete, it's time to set up a Phoenix REST API so that you can publish and persist your sensor data to PostgreSQL+TimescaleDB. Once you have an HTTP server up and running, you'll revisit your Nerves application and add a data publisher subproject to the poncho project, similarly to how you created the `veml6030` subproject.