Extracted from:

Build a Weather Station with Elixir and Nerves

Visualize Your Sensor Data with Phoenix and Grafana

The Pragmatic Bookshelf

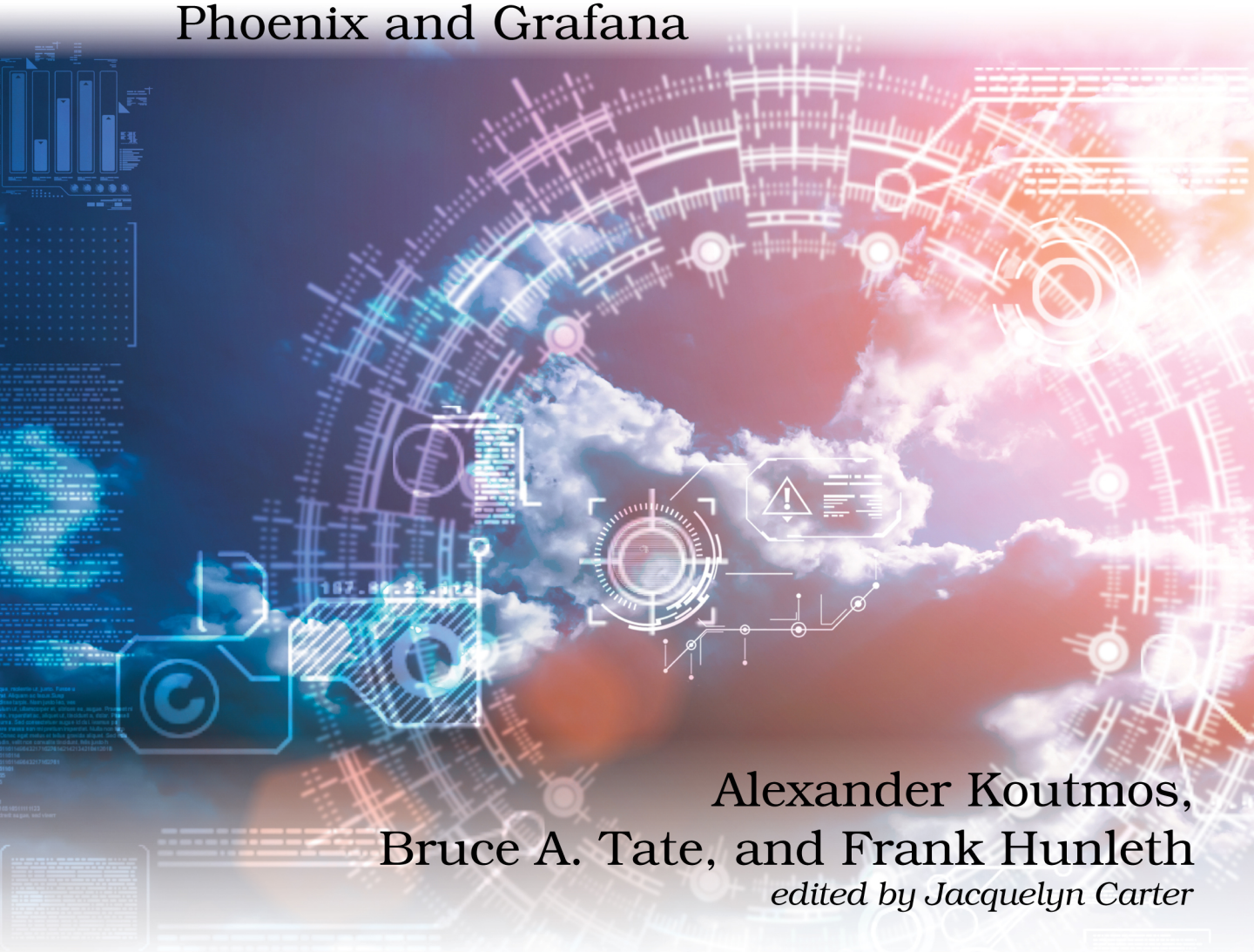
Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Build a Weather Station with Elixir and Nerves

Visualize Your Sensor Data with Phoenix and Grafana



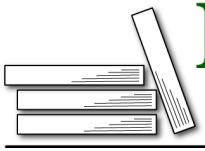
Alexander Koutmos,
Bruce A. Tate, and Frank Hunleth
edited by Jacquelyn Carter

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

Time-Series Sensor Hub

To experience the robustness of Elixir in an IoT setting, you'll be leveraging the Nerves framework and its many tools to build a sensor hub weather station. The IoT sensor hub will collect weather data at a regular interval and then publish that data to a Phoenix RESTful API. So that you can retrieve this weather data for later review, your Phoenix server will be persisting that sensor weather data into PostgreSQL.

While we could use a vanilla install of PostgreSQL for this project, the nature of our data is time-series, and it would be best if we leverage a time-series database. Specifically, it will be far more performant if we queried our sensor data from a persistent datastore that supports time-series data as a first class citizen. Luckily for us, a PostgreSQL extension solves this exact problem and it's called TimescaleDB.

By leveraging the TimescaleDB extension, you get all the benefits of using PostgreSQL as well as utilities for dealing specifically with time-series data. Under the hood, the TimescaleDB extension will automatically partition your data by time and allow you to interact with this data as if it were all contained within one database table. The database table that you interact with is also known as a hypertable and is merely a facade for all of the time-sliced partitions.[1] This takes all the administrative overhead out of manually partitioning tables and creating new partitions as days/weeks roll over in the database. From the Elixir side of things, given that we are still interacting with a PostgreSQL database, all of our database interactions still take place using Ecto.

As you'll see from working on the project throughout this book, the pairing of a time-series database with an IoT sensor hub is a very powerful technology stack. For this particular project, we'll be connecting to a wireless LAN and will be publishing the measurements to a server running on the LAN. While in a real-world setting these IoT devices may find themselves in remote locations well out of reach of a WiFi network, the proposed setup is suitable for this application.

If you're interested in building Nerves projects that run in remote environments, be sure to check out the VintageNetMobile[2] and VintageNetQMI[3] projects to see how you can leverage cellular modems from your embedded devices.

1. https://docs.timescale.com/latest/introduction/architecture
2. https://github.com/nerves-networking/vintage_net_mobile
3. https://github.com/nerves-networking/vintage_net_qmi

Alex says:

Partitioning by Space

In addition to partitioning your data efficiently by time, TimescaleDB also allows you to partition your data by space. What this means is that you can tell TimescaleDB to further partition each hypertable by an additional dimension to extract even more performance out of TimescaleDB. This additional dimension is derived from your table's schema and can be any other column aside from the timestamp column that you used to create the hypertable.

Leveraging space partitioning can be useful when you have a dataset that is well suited for time and space partitioning, but it can also decrease performance if the dataset isn't well suited.[a] In the context of IoT applications, it may be useful to space partition inbound data by the device that it was published from or by a geographical region where multiple devices have been deployed to.

a. https://docs.timescale.com/timescaledb/latest/how-to-guides/hypertables/best-practices/#space-partitions

Laying Out the Architecture

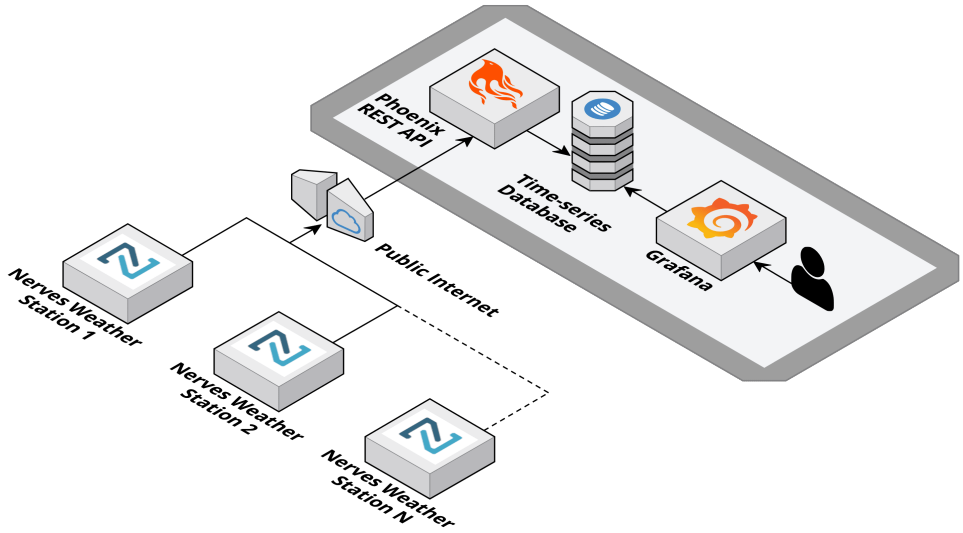
Given that there are a number of components to this project (a time-series database, a nerves sensor hub, and a Phoenix back-end API) and they all operate at different layers, it would be beneficial to first visualize all the parts and how they interact with one another. Let's take a look at the architecture diagram on page 3 and break down how the various components will work with one another.

At the bottom left of the diagram you'll notice that we have entries for Nerves Weather Stations 1, 2, and N, which denotes that we can arbitrarily scale our IoT fleet up and down as the need arises. In a real-world application, you may have a vast number of sensor hubs deployed, all of which are reporting back to your server-side application. Thus the Phoenix API acts as the gateway for all of the sensor data collected by your IoT devices.

Between our fleet of IoT sensor hubs and our Phoenix API is a network interface. For the weather station application that we'll be building throughout this book, that network interface is the WiFi antenna built into the Raspberry Pi, and our home LAN. For production use, this network interface could be ethernet or LTE,[4] depending on where your IoT device is deployed to.

Once our data is pushed from our Nerves IoT devices to our Phoenix API (via HTTP), our server-side application will persist that data into our TimescaleDB-

4. https://github.com/nerves-networking/vintage_net_mobile



enabled PostgreSQL instance. One of the reasons that TimescaleDB is a good fit for the problem at hand is that it deals well with high-cardinality data.[5]

What Is Cardinality?

Cardinality, as it pertains to the data stored in a database, is a measure of how many different values are present for a particular field or column. For example, if you're using UUIDs to capture a user's `id`, you will have high cardinality because each user will have a unique ID. In other words, if you have 50,000 users, you will have 50,000 possible values for the `id` column.

On the other hand, if you have have a column called `user_type`, for example, and that is an enum with possible values of `basic`, `admin`, and `super_user`, you would have low cardinality for the `user_type` column. The reason for this is that even if you have 50,000 users stored in your database table, `user_type` can only be one of three possible values.

After our Phoenix application has persisted our data into PostgreSQL, its responsibilities with regards to that data are effectively over. We could possibly extend our Phoenix application to return this data via a RESTful API or even present the time-series data via a LiveView SVG chart.[6] But in the spirit of

5. https://blog.timescale.com/blog/what-is-high-cardinality-how-do-time-series-databases-influxdb-timescaledb-compare
6. https://github.com/mindok/contex

getting things up and running quickly, we'll lean on the powerful yet simple data visualization tool Grafana.[7]

Grafana has the ability to connect to a wide array of data sources like Prometheus,[8] InfluxDB,[9] and even PostgreSQL+TimescaleDB.[10] This will allow us to host our own instance of Grafana (via Docker), which we can then interact with, to visualize all of our time-series data as it is persisted into our database.

With a high-level understanding of how all the pieces fit together and how they communicate with one another, it's time to dive deeper into the Nerves side of things and see how you'll be structuring your Nerves project.

7. https://grafana.com/
8. https://prometheus.io
9. https://www.influxdata.com
10. https://grafana.com/docs/grafana/latest/datasources/postgres/