MIGUEL COBÁ

DEPLOYING ELIXIR

ADVANCED TOPICS

Deploy to AWS, Azure, and GCP. Create clusters and orchestrate them with Kubernetes

Table of Contents

About this book 1

Preface 2

Prerequisites 3

- macOS 3
- Homebrew 3
- asdf 3
- Erlang and Elixir 3
- Phoenix 4
- PostgreSQL 4

Elixir/Phoenix sample application 5

- Create a Phoenix application 5
- Run the application using `mix` 5
- Run the application using Elixir Releases 6
- Run the application using Docker 8

Elixir clustering with `libcluster` 9

- Nodes and Clusters 9
- Clustering in Erlang 9
- Clustering in Elixir 10
- Clustering with `libcluster`. 13
- Cluster.Strategy.Epmd Strategy 14
- Cluster.Strategy.LocalEpmd Strategy 17
- Cluster.Strategy.ErlangHosts Strategy 17
- Cluster.Strategy.Gossip Strategy 17
- `libcluster` in Kubernetes 17

PostgreSQL on Kubernetes 19

- Prerequisites 19
- PostgreSQL and Docker 19
- PostgreSQL and Kubernetes 22

Deploying Elixir to a Kubernetes cluster 34

- Kubernetes 34
- Container Registry 34
- Cluster.Strategy.Kubernetes.DNS Strategy 35
- Publish our Elixir Docker image to Docker Hub 37
- Kubernetes configuration 39
- Scaling your application 48
- Kubernetes ConfigMaps and Secrets 49
- Adding a Secret to our Kubernetes cluster 50

Deploying to Microsoft Azure 57

- Azure Kubernetes Service (AKS) . . . 57
- Azure Resources and Resources Groups . . . 57
- Create a Container Registry. . . . 58
- Create a Kubernetes cluster . . . 60
- Connect to Kubernetes cluster with kubectl . . . 62
- Deploy our PostgreSQL database to AKS . . . 63
- Create a Secret on AKS . . . 65
- Deploy our Application to AKS . . . 66
- Scaling the AKS cluster . . . 70
- Deleting the Resource Group . . . 71

Deploying to Amazon Web Services . . . 73

- Elastic Kubernetes Service (EKS) . . . 73
- Elastic Container Registry (ECR) . . . 73
- Create a Kubernetes cluster in ECR . . . 75
- Connect to Kubernetes cluster with kubectl . . . 76
- Deploy our PostgreSQL database to EKS . . . 76
- Create a Secret on EKS . . . 79
- Deploy our Application to EKS . . . 79
- Scaling the EKS cluster . . . 84
- Deleting your EKS cluster . . . 85

Deploying to Google Cloud Platform . . . 87

- Google Kubernetes Engine (GKE) . . . 87
- Authenticate `gcloud` to Google Cloud . . . 87
- Google Cloud Projects . . . 89
- Google Cloud Billing Account . . . 90
- Google Artifact Registry . . . 91
- Create a Kubernetes cluster in GKE . . . 94
- Connect to Kubernetes cluster with kubectl . . . 96
- Deploy our PostgreSQL database to GKE . . . 97
- Create a Secret on GKE . . . 99
- Deploy the Application to GKE . . . 99
- Scaling the GKE cluster . . . 103
- Deleting your GKE cluster . . . 104

Afterword . . . 106

Appendix A: Source Code . . . 107

Appendix B: Install Docker . . . 108

- Install Docker Desktop with `brew` . . . 108
- Start Docker Desktop . . . 108
- Test Docker CLI . . . 108

Install Minikube . . . 109

- Install Minikube with `brew` . . . 109

Test Docker CLI 109
Signup for Docker Hub 109
Appendix C: Microsoft Azure 112
Signup for Microsoft Azure 112
Install the Azure CLI 115
Login with the Azure CLI 115
Appendix D: Amazon Web Services 117
Signup for Amazon Web Services 117
Create an IAM user 125
Install the AWS CLI 131
Install `eksctl` 132
Appendix E: Google Cloud Platform (GCP) 134
Signup for Google Cloud Platform 134
Google Cloud concepts 141
Install the Google Cloud SKD 142

About this book

Deploying Elixir: Advanced Topics by *Miguel Cobá*

https://www.miguelcoba.com

This book was written in Visual Studio Code with Asciidoctor on macOS Monterey.

1st Edition

Version: v1.0

Preface

This book is the second part of "Deploying Elixir", a book I wrote in 2021.

"Deploying Elixir" was downloaded by more than 1000 developers all over the world and many of those approached to me with requests to write about topics not covered in it. Those topics were more advanced and often encountered in professional environments.

After realizing that there was enough interest in the Elixir community to learn about deploying Elixir in professional environments, I set a plan to write about it.

I present to you **"Deploying Elixir: Advanced Topics"** a book where I'll show you how to deploy Elixir using technologies used frequently in Enterprises and Startups.

My goal is that, after reading this book, you'll have a solid understanding of Elixir clustering, Kubernetes clusters and orchestration, and have practice deploying Elixir projects in Cloud Platforms like Amazon Web Services, Microsoft Azure, and Google Cloud Platform.

Elixir clustering with `libcluster`

In this chapter I'll show you how to create an Elixir cluster.

Nodes and Clusters

Normally, when we create an Elixir application with `mix phx.new`, we start it using `mix phx.server` and access it by opening a browser to http://localhost:4000. We see our application in the browser and everything just works.

But, just below everything, is the Erlang Virtual Machine (VM) running our application and taking care of making it work.

An running instance of the Erlang Virtual Machine is called a **Node**.

When we run our Elixir app, we are running it in a single Erlang node.

If the Node dies for some reason, all the process that the Erlang Virtual Machine is managing will die. In this example, if our Node dies, our Elixir application dies too. For some applications this is not convenient and better availability is desired.

One way of increasing the availability of an application is by running several copies at the same time, so that if one dies, the others can still provide the functionality the application provides.

A collection of nodes connected and working together is called a **Cluster**.

The Erlang VM has native support to connect nodes so that they are aware of each other and can communicate among themselves.

It is extremely simple to connect Erlang nodes.

Clustering in Erlang

Let's see a quick example of connecting Erlang Nodes. Open two shell terminals.

In the first one write `erl -sname node1`. You'll see this:

```
erl -sname node1
Erlang/OTP 24 [erts-12.3.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]
[jit]

Eshell V12.3.1  (abort with ^G)
(node1@smaug)1>
```

You can see that the prompt shows the node name in the form `name@host`. In my case, my laptop is called `smaug`. The name is the value you passed to the `-sname` argument to `erl`. Your prompt will show your computer name instead.

In the second terminal write `erl -sname node2`. You'll see this:

```
erl -sname node2
Erlang/OTP 24 [erts-12.3.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]
[jit]

Eshell V12.3.1  (abort with ^G)
(node2@smaug)1>
```

You have now two instances of the Erlang Virtual Machine running on your computer. That means that two different nodes are running now. But they don't know each other.

Let's connect them. In the **first** terminal write this (replacing the name for **your** second node's name):

```
(node1@smaug)1> net_adm:ping('node2@smaug').
pong
```

If you now write `nodes().` on both nodes, you'll see that the nodes now know each other.

First node:

```
(node1@smaug)2> nodes().
[node2@smaug]
```

Second node:

```
(node2@smaug)1> nodes().
[node1@smaug]
```

This, technically, is your first Erlang Cluster. Yay!

Clustering in Elixir

In Elixir the process is very similar. Open two shell sessions.

In the first one write, `iex --sname node1`. You'll see this:

```
iex --sname node1
Erlang/OTP 24 [erts-12.3.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]
[jit]

Interactive Elixir (1.13.4) - press Ctrl+C to exit (type h ENTER for help)
iex(node1@smaug)1>
```

In the second one write, `iex --sname node2`. You'll see this:

```
iex --sname node2
Erlang/OTP 24 [erts-12.3.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]
[jit]

Interactive Elixir (1.13.4) - press Ctrl+C to exit (type h ENTER for help)
iex(node2@smaug)1>
```

Ok, now we have two nodes each one running Elixir on top of the Erlang Virtual Machine. Let's connect them.

In the first terminal write `Node.connect(:node2@smaug)`. As you see the command now uses Elixir syntax. You should see this:

```
iex(node1@smaug)1> Node.connect(:node2@smaug)
true
```

Let's verify that the nodes are connected. Run `Node.list()` on both nodes.

First node:

```
iex(node1@smaug)2> Node.list()
[:node2@smaug]
```

Second node:

```
iex(node2@smaug)1> Node.list()
[:node1@smaug]
```

Now you have an Elixir cluster composed of two nodes.

Showing node information with Phoenix

Before continuing to more advanced examples let's modify our Phoenix application to show the node and cluster information directly in the browser.

Open your `lib/neptune_web/controllers/page_controller.ex` file and change the `index/2` function to be like this:

lib/neptune_web/controllers/page_controller.ex

```
def index(conn, _params) do
  this_node = node()
  other_nodes = Node.list()
  render(conn, "index.html", %{this_node: this_node, other_nodes: other_nodes})
end
```

This function is getting the node and cluster details from the Erlang Virtual Machine where our Phoenix application is running and passing them to the `index.html` template to be rendered.

Now modify `lib/neptune_web/templates/page/index.html.heex` and change its contents to these:

lib/neptune_web/templates/page/index.html.heex

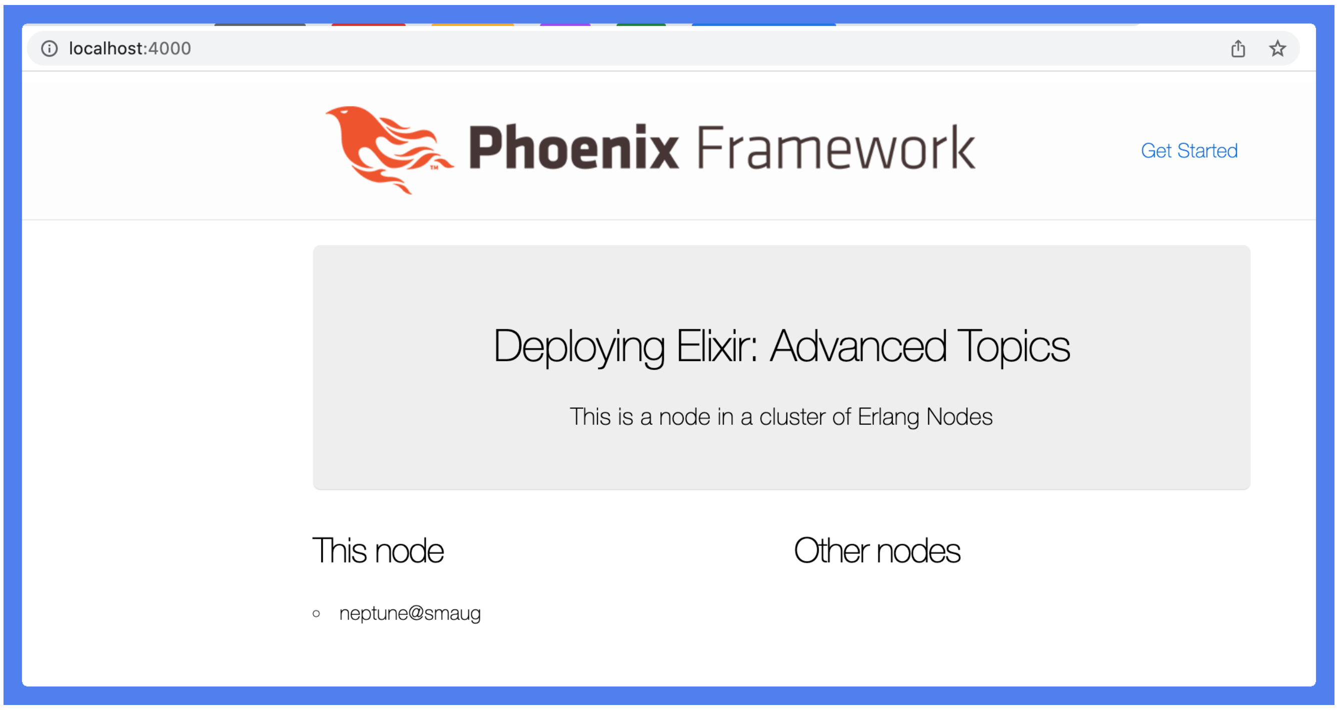
```
<section class="phx-hero">
  <h1>Deploying Elixir: Advanced Topics</h1>
  <p>This is a node in a cluster of Erlang Nodes</p>
</section>

<section class="row">
  <article class="column">
    <h2>This node</h2>
    <ul>
      <li>
        <%= @this_node %>
      </li>
    </ul>
  </article>
  <article class="column">
    <h2>Other nodes</h2>
    <ul>
      <%= for node <- @other_nodes do %>
        <li>
          <%= node %>
        </li>
      <% end %>
    </ul>
  </article>
</section>
```

Now:

- refresh the page if you're using `mix phx.server`
- rebuild the release with `MIX_ENV=prod mix release --overwrite` if you're using Elixir Release
- rebuild the Docker image with `docker build -t neptune .` if you're using Docker

The app will now show the cluster nodes' details on it:



Deploying Elixir: Advanced Topics

This is a node in a cluster of Erlang Nodes

This node

- neptune@smaug

Other nodes

The source code for this section is on the `elixir-nodes-info` branch.

Clustering with `libcluster`.

As an exercise, it is good to know how to connect nodes manually to create a cluster, but this is clearly not something you'll do on production. Nobody is going to login to some remote server everytime to open a `iex` session and connect manually all the nodes in the cluster.

This needs to be automated so that the cluster forms itself correctly without human intervention.

To solve this problem, Paul Schoenfelder created the `libcluster` library to automatically form clusters of Erlang nodes.

This library has several ways of forming clusters depending on which strategy you select for determining nodes membership to the cluster.

Let's configure our sample app to use `libcluster`.

Add the `libcluster` dependency to your `mix.exs`:

```
defp deps do
  [
    # ...
    {:plug_cowboy, "~> 2.5"},
    {:libcluster, "~> 3.3"}
  ]
end
```

And download it with `mix deps.get`. Then modify your `application.ex` and add the `Cluster.Supervisor` to the list of children:

```
  def start(_type, _args) do
    topologies = Application.get_env(:libcluster, :topologies) || []

    children = [
      Neptune.Repo,
      NeptuneWeb.Telemetry,
      {Phoenix.PubSub, name: Neptune.PubSub},
      NeptuneWeb.Endpoint,
      {Cluster.Supervisor, [topologies, [name: Neptune.ClusterSupervisor]]}
    ]

    opts = [strategy: :one_for_one, name: Neptune.Supervisor]
    Supervisor.start_link(children, opts)
  end
```

We are ready to use `libcluster` to create our Elixir cluster. But first we need to configure it so that it knows which nodes are going to be part of the cluster.

`libcluster` has several strategies to do this. Let's try each of those.

You can find the source code for this section in the branch `elixir-libcluster`.

Cluster.Strategy.Epmd Strategy

This strategy relies on `epmd` (Erlang Port Mapper Daemon) to form a cluster from a **configured** set of nodes.

`epmd` is started automatically on a computer if a node is started in distributed mode (meaning that it was started with the -sname or -name parameters) and no instance of the daemon exists. You don't need to worry about it. It will be there.

This strategy requires us to explicitly list all the nodes we want to be part of the cluster. Let's do it.

Open `config.exs` and add this just before the last `import_config` line:

```
config :libcluster,
  topologies: [
    epmd_example: [
      strategy: Elixir.Cluster.Strategy.Epmd,
      config: [
        hosts: [:node4000@smaug, :node4001@smaug]
      ]
    ]
  ]
```

Now we can start two nodes with the names `node4000` and `node4001` and `libcluster` will ensure to form a cluster with them.

A couple of considerations.

Normally we start the app with `mix phx.server`, but here we need to specify the node name for each node we start so instead of using `mix phx.server` we are going to start the app with `iex --sname <NODE_NAME> -S mix phx.server`.

In dev mode, port `4000` is hardcoded in `config/dev.exs`. To start two instances of our application we could change the port manually before starting a each node. A better way is to start the application in `prod` mode taking advantage of the `config/runtime.exs` configuration that reads the port to start on from an environment variable.

Let’s take the second approach and start the nodes in `prod` mode and setting the required environment variables.

Start the first node:

```
export SECRET_KEY_BASE=$(mix phx.gen.secret)
export PORT=4000
export PHX_SERVER=true
export DATABASE_URL=ecto://postgres:postgres@localhost/neptune_prod
export MIX_ENV=prod
iex --sname node4000 -S mix phx.server
```

You should see this a warning saying that can’t connect to `node4001@smaug`. That’s expected as we haven’t started the second node:

```
Erlang/OTP 24 [erts-12.3.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]
[jit]

Generated neptune app
03:38:18.221 [info] Running NeptuneWeb.Endpoint with cowboy 2.9.0 at :::4000 (http)
03:38:18.224 [info] Access NeptuneWeb.Endpoint at http://example.com:443
03:38:18.239 [warning] [libcluster:epmd_example] unable to connect to :node4001@smaug
Interactive Elixir (1.13.4) - press Ctrl+C to exit (type h ENTER for help)
iex(node4000@smaug)1>
```

Start the second node:

```
export SECRET_KEY_BASE=$(mix phx.gen.secret)
export PORT=4001
export PHX_SERVER=true
export DATABASE_URL=ecto://postgres:postgres@localhost/neptune_prod
export MIX_ENV=prod
iex --sname node4001 -S mix phx.server
```

You should see this:

```
Erlang/OTP 24 [erts-12.3.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]
[jit]

03:38:46.848 [info] Running NeptuneWeb.Endpoint with cowboy 2.9.0 at :::4001 (http)
03:38:46.854 [info] Access NeptuneWeb.Endpoint at http://example.com:443
03:38:46.872 [info] [libcluster:epmd_example] connected to :node4000@smaug
Interactive Elixir (1.13.4) - press Ctrl+C to exit (type h ENTER for help)
iex(node4001@smaug)1>
```

Now run `Node.list()` in both nodes to confirm that they are connected.

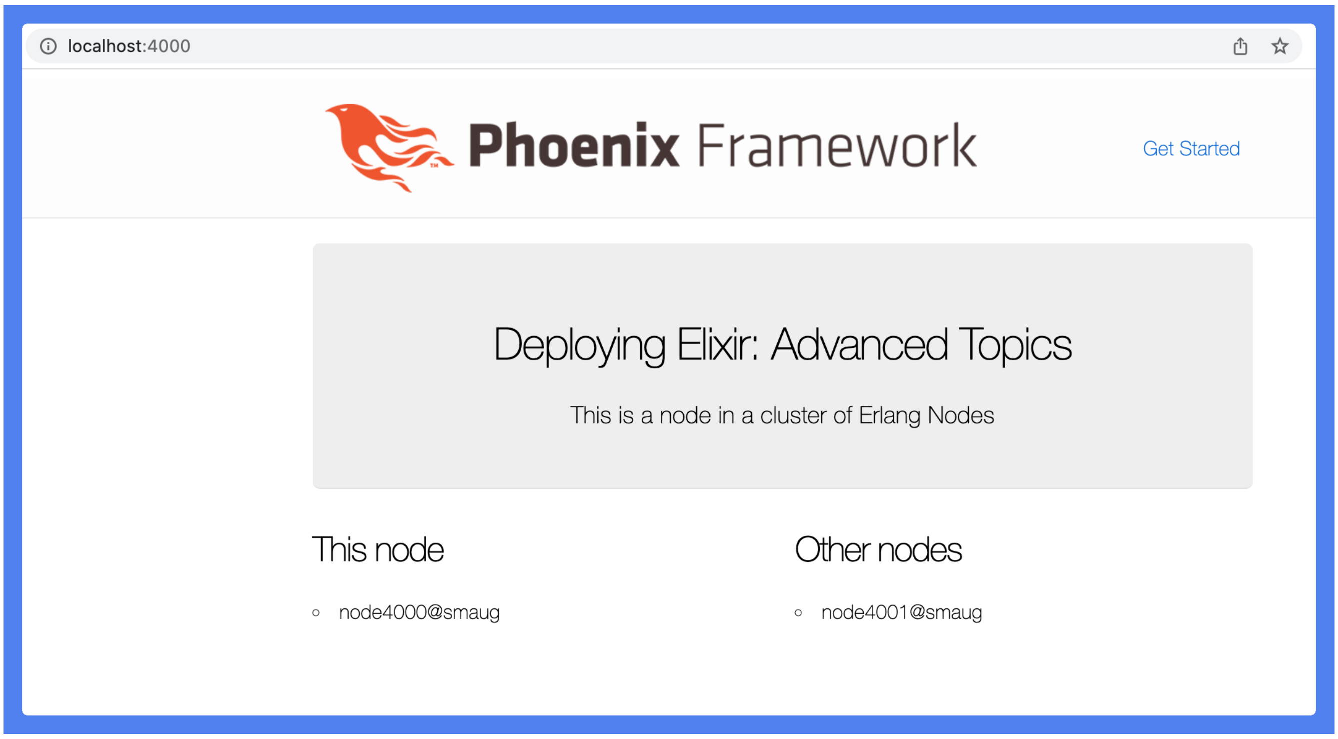
In the first node:

```
iex(node4000@smaug)1> Node.list()
[:node4001@smaug]
```

In the second node:

```
iex(node4001@smaug)1> Node.list()
[:node4000@smaug]
```

If you open http://localhost:4000 you'll see the sample app running in the first node:



Deploying Elixir: Advanced Topics

This is a node in a cluster of Erlang Nodes

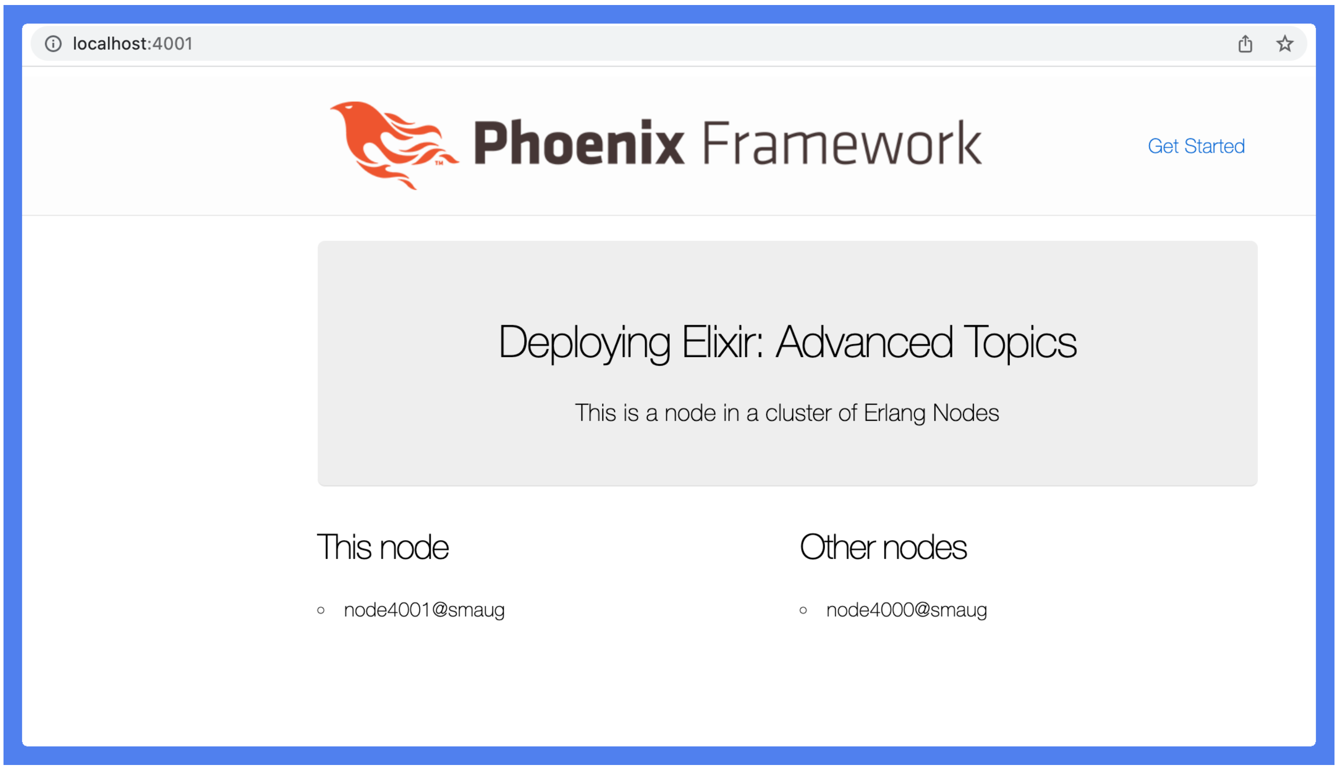
This node

- node4000@smaug

Other nodes

- node4001@smaug

And if you open http://localhost:4001 you'll see the sample app running in the second node:



Deploying Elixir: Advanced Topics

This is a node in a cluster of Erlang Nodes

This node

- node4001@smaug

Other nodes

- node4000@smaug

Good. We have a cluster with two Elixir nodes.

The main disadvantage of this strategy is that we need to hardcode all the names of the nodes that we want to include in the cluster.

You can find the source code for this section in the branch `elixir-libcluster-epmd`

Cluster.Strategy.LocalEpmd Strategy

Get the full book to read this section

Cluster.Strategy.ErlangHosts Strategy

Get the full book to read this section

Cluster.Strategy.Gossip Strategy

Get the full book to read this section

`libcluster` in Kubernetes

The remaining strategy I want to discuss is the one that allows us to create an Erlang cluster inside a Kubernetes cluster.

Before we can deploy our Elixir application to Kubernetes we need to see how we can provide our application with a database.

In the next chapter we are going to see how we can deploy a PostgreSQL server to Kubernetes.

When we finally deploy our Elixir cluster we'll configure it to use this database.