GIGALIXIR Documentation

Release 1.1.3

Invisible Software, Inc.

Feb 13, 2020

Contents:

Gigalixir is Elixir's Platform-as-a-Service. For more information, see https://gigalixir.com. Get 1 instance + 1 database for free without a credit card.

CHAPTER 1

What is Gigalixir?

Gigalixir is a fully-featured, production-stable platform-as-a-service built just for Elixir that saves you money and unlocks the full power of Elixir and Phoenix without forcing you to build production infrastructure or deal with maintenance and operations. For more information, see https://gigalixir.com.

Try Gigalixir for free without a credit card by following the *Getting Started Guide*.

CHAPTER 2

Screencast

Note: The video is out of date, so you can't follow it verbatim, but the steps are the same.

For those of you who prefer a screencast introduction. Elixircasts.io has a great video that they've generously allowed us to embed here.

CHAPTER 3

Getting Started Guide

The goal of this guide is to get your app up and running on Gigalixir. You will sign up for an account, prepare your app, deploy, and provision a database.

If you're deploying an open source project, we provide consulting services free of charge. *Contact us* and we'll send you a pull request with everything you need to get started.

3.1 Prerequisites

1. `python3`. `python2` also works, but it is EOL as of January 1st, 2020.
2. `pip3`. For help, take a look at the pip documentation.
3. `git`. For help, take a look at the git documentation.
4. Linux, OS X, or Windows (beta).

For example, on Ubuntu, run

```
sudo apt-get update
sudo apt-get install -y python3 python3-pip git-core curl
```

3.2 Install the Command-Line Interface

Next, install the command-line interface. Gigalixir has a web interface at https://gigalixir.com/#/dashboard, but you will still need the CLI to do anything other than signup, deploy, and scale.

```
pip3 install gigalixir --ignore-installed six
```

Make sure the executable is in your path, if it isn't already.

```
echo 'export PATH=~/.local/bin:$PATH' >> ~/.bash_profile
```

Reload the profile

```
source ~/.bash_profile
```

Verify by running

```
gigalixir --help
```

The reason we ignore six is because OS X has a pre-installed version of six that is incompatible. When pip tries to upgrade it, OS X won't let us. For more, see https://github.com/pypa/pip/issues/3165

3.3 Create an Account

If you already have an account, skip this step.

Create an account using the following command. It will prompt you for your email address and password. You will have to confirm your email before continuing. Gigalixir's free tier does not require a credit card, but you will be limited to 1 instance with 0.2GB of memory and 1 postgresql database limited to 10,000 rows.

```
gigalixir signup
```

3.4 Log In

Next, log in. This will grant you an api key. It will also optionally modify your ~/.netrc file so that all future commands are authenticated.

```
gigalixir login
```

Verify by running

```
gigalixir account
```

3.5 Prepare Your App

You *might* be able to skip this step and "just deploy", but it depends on what version of phoenix you're running and whether you are okay running in mix mode without distillery or elixir releases.

For more information, click here: *Modifying an Existing App to Run on Gigalixir*.

Or if you just want to give gigalixir a spin, clone our reference app.

```
git clone https://github.com/gigalixir/gigalixir-getting-started.git
```

3.6 Set Up App for Deploys

To create your app, run the following command. It will also set up a git remote. This must be run from within a git repository folder. An app name will be generated for you, but you can also optionally supply an app name if you wish using `gigalixir create -n $APP_NAME`. There is currently no way to change your app name once it is created. If you like, you can also choose which cloud provider and region using the `--cloud` and `--region`

options. We currently support `gcp` in `v2018-us-central1` or `europe-west1` and `aws` in `us-east-1` or `us-west-2`.

```
cd gigalixir-getting-started
APP_NAME=$(gigalixir create)
```

Verify that the app was created, by running

```
gigalixir apps
```

Verify that a git remote was created by running

```
git remote -v
```

3.7 Specify Versions

The default Elixir version is defined here which is 1.5.3 as of this writing. If you are using Phoenix 1.4 or higher, you may need to use a higher version of Elixir. Supported Elixir and erlang versions can be found at https://github.com/HashNuke/heroku-buildpack-elixir#version-support

Create a file `elixir_buildpack.config` at the root of your repo and add something like this. Make sure it matches what you have in development to ensure a smooth deploy.

```
elixir_version=1.10.0
erlang_version=22.2
```

The latest versions of phoenix also require higher versions of node. Create a file called `phoenix_static_buildpack.config` with something like

```
node_version=11.1.0
```

Don't forget to commit

```
git add elixir_buildpack.config phoenix_static_buildpack.config
git commit -m "set elixir, erlang, and node version"
```

3.8 Provision a Database

Phoenix 1.4 enforces the DATABASE_URL env var at compile time so let's create a database first, before deploying.

```
gigalixir pg:create --free
```

Verify by running

```
gigalixir pg
```

Once the database is created, verify your configuration includes a `DATABASE_URL` by running

```
gigalixir config
```

3.9 Deploy!

Finally, build and deploy.

```
git push gigalixir master
```

Wait a minute or two for the app to pass health checks. You can check the status by running

```
gigalixir ps
```

Once it's healthy, verify it works

```
curl https://$APP_NAME.gigalixirapp.com/
# or you could also run
# gigalixir open
```

3.10 Run Migrations

If you are not using releases, the easiest way to run migrations is as a job.

```
gigalixir run mix ecto.migrate
# this is run asynchronously as a job, so to see the progress, you need to run
gigalixir logs
```

If you are using distillery or Elixir releases, your app needs to be up and running, then run

```
# pg:migrate runs migrations from your app node so you need to ssh in to run it
# we need to add ssh keys first
gigalixir account:ssh_keys:add "$(cat ~/.ssh/id_rsa.pub)"
gigalixir ps:migrate
```

For more, see *How to Run Migrations*.

3.11 What's Next?

- *How to Configure an App*
- *How to Check App Status*
- *How to Tail Logs*
- *How to Scale an App*
- *How to Restart an App*
- *How to Rollback an App*
- *How to Drop into a Remote Console*
- *How to Launch a Remote Observer*
- *How to Hot Upgrade an App*

CHAPTER 4

Modifying an Existing App to Run on Gigalixir

Whether you have an existing app or you just ran `mix phx.new`, the goal of this guide is to get your app ready for deployment on Gigalixir. We assume that you are using Phoenix here. If you aren't, feel free to *contact us* for help. As long as your app is serving HTTP traffic on `$PORT`, you should be fine.

Important: If you have an umbrella app, be sure to *also* see *How do I deploy an umbrella app?*.

4.1 Mix vs Distillery vs Elixir Releases

It's typically recommended to use distillery when you're ready to deploy, but if you prefer, you can also use plain mix or Elixir releases (new in Elixir 1.9).

You're probably already used to mix from development and deploying with mix is simpler and easier, but you can't do hot upgrades, clustering, remote observer, and maybe a few other things.

Elixir releases is still very new and doesn't support hot upgrades, but it is built-in to Elixir so you don't have to add an extra dependency such as distillery to get clustering, remote console, observer, etc.

If you deploy with distillery, you no longer get mix tasks like `mix ecto.migrate` and configuring your `prod.exs` can be confusing in some cases.

If you don't know which to choose, we generally recommend going with distillery because.. why use Elixir if you can't use all its amazing features? Also, Gigalixir works hard to make things easy with distillery. For example, we have a special command, `gigalixir ps:migrate`, that makes it easy to run migrations without mix.

If you choose mix, see *Using Mix*.

If you choose distillery, see *Using Distillery*.

If you choose Elixir releases, see *Using Elixir Releases*.

4.2 Using Mix

For an example app that uses mix and works on gigalixir, see https://github.com/gigalixir/gigalixir-getting-started/tree/js/mix

4.2.1 Configuration and Secrets

As of Phoenix 1.4.4+, `prod.secret.exs` has been modernized and uses environment variables for configuration which is exactly what we want. If you plan to use this and are on a free-tier database, make sure that you either set the `POOL_SIZE` environment variable by running `gigalixir config:set POOL_SIZE=2` or change the default value in `prod.secret.exs` to "2". If you are running an older version of Phoenix, you'll probably want to delete your `prod.secret.exs` file, and comment out the line in your `prod.exs` that imports it.

Then append something like the following in `prod.exs`. Don't replace what you already have, just add this to the bottom.

```
config :gigalixir_getting_started, GigalixirGettingStartedWeb.Endpoint,
  http: [port: {:system, "PORT"}], # Possibly not needed, but doesn't hurt
  url: [host: System.get_env("APP_NAME") <> ".gigalixirapp.com", port: 80],
  secret_key_base: Map.fetch!(System.get_env(), "SECRET_KEY_BASE"),
  server: true

config :gigalixir_getting_started, GigalixirGettingStarted.Repo,
  adapter: Ecto.Adapters.Postgres,
  url: System.get_env("DATABASE_URL"),
  ssl: true,
  pool_size: 2 # Free tier db only allows 4 connections. Rolling deploys need pool_
↪size*(n+1) connections where n is the number of app replicas.
```

1. Replace `:gigalixir_getting_started` with your app name e.g. `:my_app`
2. Replace `GigalixirGettingStartedWeb.Endpoint` with your endpoint module name. You can find your endpoint module name by running something like

   ```
   grep -R "defmodule.*Endpoint" lib/
   ```

   Phoenix 1.2, 1.3, and 1.4 give different names so this is a common source of errors.
3. Replace `GigalixirGettingStarted.Repo` with your repo module name e.g. `MyApp.Repo`

You don't have to worry about setting your `SECRET_KEY_BASE` config because we generate one and set it for you.

Don't forget to commit your changes

```
git add config/prod.exs
git commit -m "setup production deploys"
```

4.2.2 Verify

Let's make sure everything works.

```
APP_NAME=foo SECRET_KEY_BASE="$(mix phx.gen.secret)" MIX_ENV=prod DATABASE_URL=
↪"postgresql://user:pass@localhost:5432/foo" PORT=4000 mix phx.server
```

Check it out.

```
curl localhost:4000
```

If everything works, continue to *Set Up App for Deploys*.

4.2.3 Specify Buildpacks (optional)

We rely on buildpacks to compile and build your release. We auto-detect a variety of buildpacks so you probably don't need this, but if you want to specify your own buildpacks create a `.buildpacks` file with the buildpacks you want. For example,

```
https://github.com/HashNuke/heroku-buildpack-elixir
https://github.com/gjaldon/heroku-buildpack-phoenix-static
https://github.com/gigalixir/gigalixir-buildpack-mix.git
```

`heroku-buildpack-phoenix-static` is optional if you do not have Phoenix static assets. For more information about buildpacks, see *Life of a Deploy*.

Note, that the command that gets run in production depends on what your last buildpack is.

- If the last buildpack is `gigalixir-buildpack-mix`, then the command run will be something like `elixir --name $MY_NODE_NAME --cookie $MY_COOKIE -S mix phx.server`.
- If the last buildpack is `heroku-buildpack-phoenix-static`, then the command run will be `mix phx.server`.
- If the last buildpack is `heroku-buildpack-elixir`, then the command run will be `mix run --no-halt`.

If your command is `mix run --no-halt`, but you are running Phoenix (just not the assets pipeline), make sure you set `server: true` in `prod.exs`.

We highly recommend keeping `gigalixir-buildpack-mix` last so that your node name and cookie are set properly. Without those, remote_console, ps:migrate, observer, etc won't work.

4.3 Using Distillery

For an example app that uses distillery and works on gigalixir, see https://github.com/gigalixir/gigalixir-getting-started

4.3.1 Install Distillery to Build Releases

In short, you'll need to add something like this to the `deps` list in `mix.exs`

```
{:distillery, "~> 2.1"}
```

Important: If you are running Elixir 1.9, then you *must* use distillery 2.1 or greater. Elixir 1.9 and distillery below 2.1 both use *mix release* and Elixir's always takes precedence. Distillery 2.1 renames the task to *mix distillery.release*.

Then, run

```
mix deps.get
mix distillery.init
# if you are running distillery below version 2.1, you'll want to run `mix release.
↪init` instead
```

Don't forget to commit

```
git add mix.exs mix.lock rel/
git commit -m 'install distillery'
```

4.3.2 Configuration and Secrets

As of Phoenix 1.4.4+, `prod.secret.exs` has been modernized and uses environment variables for configuration which is exactly what we want. If you plan to use this and are on a free-tier database, make sure that you either set the `POOL_SIZE` environment variable by running `gigalixir config:set POOL_SIZE=2` or change the default value in `prod.secret.exs` to "2". If you are running an older version of Phoenix, you'll probably want to delete your `prod.secret.exs` file, and comment out the line in your `prod.exs` that imports it.

Then add something like the following in `prod.exs`

```
config :gigalixir_getting_started, GigalixirGettingStartedWeb.Endpoint,
  load_from_system_env: true,
  http: [port: {:system, "PORT"}], # Needed for Phoenix 1.2 and 1.4. Doesn't hurt for␣
↪1.3.
  server: true, # Without this line, your app will not start the web server!
  secret_key_base: "${SECRET_KEY_BASE}",
  url: [host: "${APP_NAME}.gigalixirapp.com", port: 443],
  cache_static_manifest: "priv/static/cache_manifest.json"

config :gigalixir_getting_started, GigalixirGettingStarted.Repo,
  adapter: Ecto.Adapters.Postgres,
  url: "${DATABASE_URL}",
  database: "",
  ssl: true,
  pool_size: 2 # Free tier db only allows 4 connections. Rolling deploys need pool_
↪size*(n+1) connections where n is the number of app replicas.
```

`server: true` **is very important and is commonly left out. Make sure you have this line.**

1. Replace `:gigalixir_getting_started` with your app name e.g. `:my_app`
2. Replace `GigalixirGettingStartedWeb.Endpoint` with your endpoint module name. You can find your endpoint module name by running something like

   ```
   grep -R "defmodule.*Endpoint" lib/
   ```

   Phoenix 1.2, 1.3, and 1.4 give different names so this is a common source of errors.
3. Replace `GigalixirGettingStarted.Repo` with your repo module name e.g. `MyApp.Repo`

You don't have to worry about setting your `SECRET_KEY_BASE` config because we generate one and set it for you. If you don't use a gigalixir managed postgres database, you'll have to set the `DATABASE_URL` yourself.

4.3.3 Verify

Let's make sure everything works.

First, try generating building static assets

```
mix deps.get

# generate static assets
cd assets
```

(continues on next page)

(continued from previous page)

```
npm install
npm run deploy
cd ..
mix phx.digest
```

and building a Distillery release locally

```
MIX_ENV=prod mix distillery.release --env=prod
# if you are running distillery below 2.1, you'll want to run this instead: MIX_
↪ENV=prod mix release --env=prod
```

and running it locally

```
MIX_ENV=prod APP_NAME=gigalixir_getting_started SECRET_KEY_BASE="$(mix phx.gen.secret)
↪" DATABASE_URL="postgresql://user:pass@localhost:5432/foo" MY_HOSTNAME=example.com␣
↪MY_COOKIE=secret REPLACE_OS_VARS=true MY_NODE_NAME=foo@127.0.0.1 PORT=4000 _build/
↪prod/rel/gigalixir_getting_started/bin/gigalixir_getting_started foreground
```

Don't forget to replace `gigalixir_getting_started` with your own app name. Also, change/add the environment variables as needed.

Commit the changes

```
git add config/prod.exs assets/package-lock.json
git commit -m 'distillery configuration'
```

Check it out.

```
curl localhost:4000
```

If that didn't work, the first place to check is `prod.exs`. Make sure you have `server: true` somewhere and there are no typos.

Also check out *Troubleshooting*.

If it still doesn't work, don't hesitate to *contact us*.

If everything works, continue to *Set Up App for Deploys*.

4.3.4 Specify Buildpacks (optional)

We rely on buildpacks to compile and build your release. We auto-detect a variety of buildpacks so you probably don't need this, but if you want to specify your own buildpacks create a `.buildpacks` file with the buildpacks you want. For example,

```
https://github.com/HashNuke/heroku-buildpack-elixir
https://github.com/gjaldon/heroku-buildpack-phoenix-static
https://github.com/gigalixir/gigalixir-buildpack-distillery.git
```

`heroku-buildpack-phoenix-static` is optional if you do not have Phoenix static assets. For more information about buildpacks, see *Life of a Deploy*.

Note, that the command that gets run in production depends on what your last buildpack is.

- If the last buildpack is `gigalixir-buildpack-distillery`, then the command run will be `/app/bin/foo foreground`.

- If the last buildpack is `heroku-buildpack-phoenix-static`, then the command run will be `mix phx.server`.
- If the last buildpack is `heroku-buildpack-elixir`, then the command run will be `mix run --no-halt`.

If your command is `mix run --no-halt`, but you are running Phoenix (just not the assets pipeline), make sure you set `server: true` in `prod.exs`.

4.3.5 Set up Node Clustering with Libcluster (optional)

If you want to cluster nodes, you should install libcluster. For more information about installing libcluster, see *Clustering Nodes*.

4.3.6 Set Up Hot Upgrades with Git v2.9.0

To run hot upgrades, you send an extra http header when running `git push gigalixir master`. Extra HTTP headers are only supported in git 2.9.0 and above so make sure you upgrade if needed. For information on how to install the latest version of git on Ubuntu, see this stackoverflow question. For information on running hot upgrades, see *How to Hot Upgrade an App* and *Life of a Hot Upgrade*.

4.4 Using Elixir Releases

4.4.1 Configuration and Secrets

Gigalixir auto-detects that you want to use Elixir Releases if you have a `config/releases.exs` file, so let's create one.

```
echo "import Config" > config/releases.exs
```

As of Phoenix 1.4.4+, `prod.secret.exs` has been modernized and uses environment variables for configuration which is exactly what we want. If you plan to use this and are on a free-tier database, make sure that you either set the `POOL_SIZE` environment variable by running `gigalixir config:set POOL_SIZE=2` or change the default value in `prod.secret.exs` to "2". If you are running an older version of Phoenix, you'll probably want to delete your `prod.secret.exs` file, and comment out the line in your `prod.exs` that imports it.

The only configuration change we really need to do now is make sure the web server is started. Add the following to your `releases.exs`.

```
config :gigalixir_getting_started, GigalixirGettingStartedWeb.Endpoint,
  server: true,
  http: [port: {:system, "PORT"}], # Needed for Phoenix 1.2 and 1.4. Doesn't hurt for␣
↪1.3.
  url: [host: System.get_env("APP_NAME") <> ".gigalixirapp.com", port: 443]
```

1. Replace `:gigalixir_getting_started` with your app name e.g. `:my_app`
2. Replace `GigalixirGettingStartedWeb.Endpoint` with your endpoint module name. You can find your endpoint module name by running something like

   ```
   grep -R "defmodule.*Endpoint" lib/
   ```

   Phoenix 1.2, 1.3, and 1.4 give different names so this is a common source of errors.

If you're using a free tier database, be sure to also set your pool size to 2 in `prod.exs`.

You don't have to worry about setting your `SECRET_KEY_BASE` config because we generate one and set it for you. If you don't use a gigalixir managed postgres database, you'll have to set the `DATABASE_URL` yourself. You can do this by running the following, but you'll need to *Install the Command-Line Interface* and login. For more information on setting configs, see *How to Configure an App*.

```
gigalixir config:set DATABASE_URL="ecto://user:pass@host:port/db"
```

4.4.2 Verify

Let's make sure everything works.

First, try generating building static assets

```
mix deps.get

# generate static assets
cd assets
npm install
npm run deploy
cd ..
mix phx.digest
```

and building a release locally

```
export SECRET_KEY_BASE="$(mix phx.gen.secret)"
export DATABASE_URL="postgresql://user:pass@localhost:5432/foo"
MIX_ENV=prod mix release
```

and running it locally

```
MIX_ENV=prod APP_NAME=gigalixir_getting_started PORT=4000 _build/prod/rel/gigalixir_
↪getting_started/bin/gigalixir_getting_started start
```

Don't forget to replace `gigalixir_getting_started` with your own app name. Also, change/add the environment variables as needed.

Check it out.

```
curl localhost:4000
```

If that didn't work, the first place to check is `prod.exs`. Make sure you have `server: true` somewhere and there are no typos.

Also check out *Troubleshooting*.

If it still doesn't work, don't hesitate to *contact us*.

If everything works, commit the changes

```
git add config/prod.exs assets/package-lock.json config/releases.exs
git commit -m 'releases configuration'
```

Continue to *Set Up App for Deploys*.

4.4.3 Specify Buildpacks (optional)

We rely on buildpacks to compile and build your release. We auto-detect a variety of buildpacks so you probably don't need this, but if you want to specify your own buildpacks create a `.buildpacks` file with the buildpacks you want. For example,

```
https://github.com/HashNuke/heroku-buildpack-elixir
https://github.com/gjaldon/heroku-buildpack-phoenix-static
https://github.com/gigalixir/gigalixir-buildpack-releases.git
```

`heroku-buildpack-phoenix-static` is optional if you do not have Phoenix static assets. For more information about buildpacks, see *Life of a Deploy*.

Note, that the command that gets run in production depends on what your last buildpack is.

- If the last buildpack is `gigalixir-buildpack-releases`, then the command run will be `/app/bin/foo start`.
- If the last buildpack is `heroku-buildpack-phoenix-static`, then the command run will be `mix phx.server`.
- If the last buildpack is `heroku-buildpack-elixir`, then the command run will be `mix run --no-halt`.

If your command is `mix run --no-halt`, but you are running Phoenix (just not the assets pipeline), make sure you set `server: true` in `prod.exs`.

CHAPTER 5

How do I install extra binaries I need for my app?

The process is different if you are using releases (distillery, Elixir releases) or mix. We recommend switching to mix mode as it's much easier. To switch to mix mode, see *How do I switch to mix mode?*.

In mix mode, all you have to do is add the relevant, buildpack to your `.buildpacks` file. Probably at the top. Make sure you also have the required Elixir, Phoenix, and mix buildpacks. For example, if you need rust installed, your `.buildpacks` file might look like this

```
https://github.com/emk/heroku-buildpack-rust
https://github.com/HashNuke/heroku-buildpack-elixir
https://github.com/gjaldon/heroku-buildpack-phoenix-static
https://github.com/gigalixir/gigalixir-buildpack-mix.git
```

For rust specifically, also be sure to run `echo "RUST_SKIP_BUILD=1" > RustConfig` since you just need the rust binaries, and don't want to build a rust project.

In mix mode, the entire build folder is packed up and shipped to your run container which means it will pack up the extra binaries you've installed and any .profile.d scripts needed to make them available. That's it!

If you want to continue using distillery, you need to manually figure out which folders and files need to be packed into your release tarball and copy them over using distillery overlays. See https://github.com/bitwalker/distillery/blob/master/docs/extensibility/overlays.md

If you are using Elixir releases, you also need to manually figure out which folders and files you need to be packed into your release tarball and copy them over using an extra "step". See https://hexdocs.pm/mix/Mix.Tasks.Release.html#module-steps

CHAPTER 6

How do I switch to mix mode?

Mix mode is sort of the default, but we automatically detect and switch you to distillery mode if you have a `rel/config.exs` file so one option is to delete that file. We also automatically detect and switch you to Elixir releases mode if you have a `config/releases.exs` file so also be sure that file is deleted.

If you don't want to delete those files, you can manually force mix mode by specifying the mix buildpack. Create a `.buildpacks` file and make sure you have something like the following. Notice that the last buildpack is the mix buildpack.

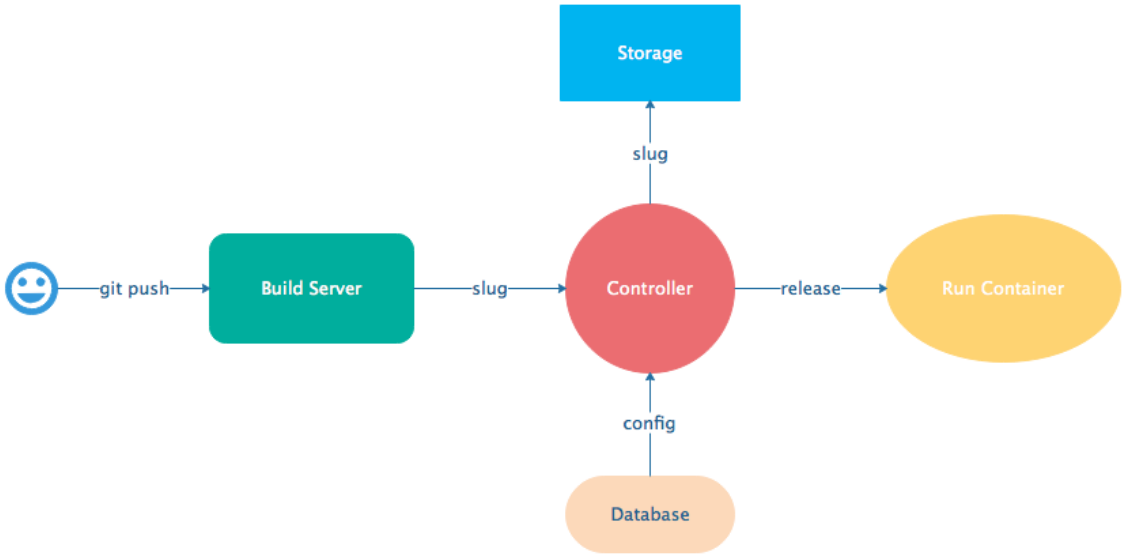
```
https://github.com/HashNuke/heroku-buildpack-elixir
https://github.com/gjaldon/heroku-buildpack-phoenix-static
https://github.com/gigalixir/gigalixir-buildpack-mix.git
```

If you wanted to force distillery or Elixir releases, you'd want the last buildpack to be either the `https://github.com/gigalixir/gigalixir-buildpack-distillery.git` or the `https://github.com/gigalixir/gigalixir-buildpack-releases.git` buildpacks, respectively.

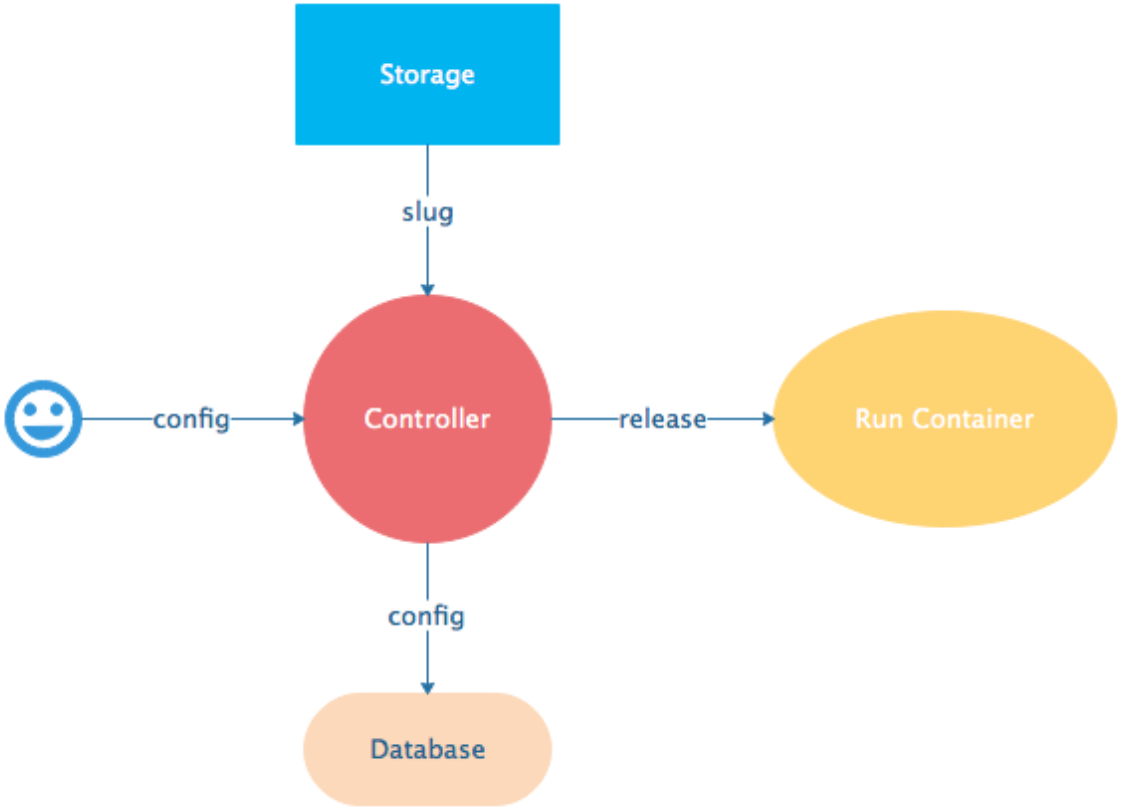
CHAPTER 7

How Does Gigalixir Work?

When you deploy an app on Gigalixir, you `git push` the source code to a build server. The build server compiles the code and assets and generates a standalone tarball we call a slug. The controller then combines the slug and your app configuration into a release. The release is deployed to run containers which actually run your app.



When you update a config, we encrypt it, store it, and combine it with the existing slug into a new release. The release is deployed to run containers.



7.1 Components

- *Build Server*: This is responsible for building your code into a release or slug.
- *API Server / Controller*: This is responsible for handling all user requests such as scaling apps, setting configs, etc. It is also responsible for deploying the release into a run container.
- *Database*: The database is where all of your app configuration is stored. Configs are encrypted due to their sensitive nature.
- *Logger*: This is responsible for collecting logs from all your containers, aggregating them, and streaming them to you.
- *Router*: This is responsible for receiving web traffic for your app, terminating TLS, and routing the traffic to your run containers.
- *TLS Manager*: This is responsible for automatically obtaining TLS certificates and storing them.
- *Kubernetes*: This is responsible for managing your run containers.
- *Slug Storage*: This is where your slugs are stored.
- *Observer*: This is an application that runs on your local machine that connects to your production node to show you everything you could ever want to know about your live production app.
- *Run Container*: This is the container that your app runs in.

- *Command-Line Interface*: This is the command-line tool that runs on your local machine that you use to control Gigalixir.

7.2 Concepts

- *User*: The user is you. When you sign up, we create a user.
- *API Key*: Every user has an API Key which is used to authenticate most API requests. You get one when you login and you can regenerate it at any time. It never expires.
- *SSH Key*: SSH keys are what we use to authenticate you when SSHing to your containers. We use them for remote observer, remote console, etc.
- *App*: An app is your Elixir application.
- *Release*: A release is a combination of a slug and a config which is deployed to a run container.
- *Slug*: Each app is compiled and built into a slug. The slug is the actual code that is run in your containers. Each app will have many slugs, one for every deploy.
- *Config*: A config is a set of key-value pairs that you use to configure your app. They are injected into your run container as environment variables.
- *Replicas*: An app can have many replicas. A replica is a single instance of your app in a single container in a single pod.
- *Custom Domain*: A custom domain is a fully qualified domain that you control which you can set up to point to your app.
- *Payment Method*: Your payment method is the credit card on file you use to pay your bill each month.
- *Permission*: A permission grants another user the ability to deploy. Even though they can deploy, you remain the owner and are responsible for paying the bill.

7.3 Life of a Deploy

When you run `git push gigalixir master`, our git server receives your source code and kicks off a build using a pre-receive hook. We build your app in an isolated docker container which ultimately produces a slug which we store for later. The buildpacks used are defined in your `.buildpacks` file.

By default, the buildpacks we use include

- https://github.com/HashNuke/heroku-buildpack-elixir.git
  - To run mix compile
  - If you want, you can configure this buildpack.
- https://github.com/gjaldon/heroku-buildpack-phoenix-static.git
  - To run mix phx.digest
  - This is only included if you have an assets folder present.
- https://github.com/gigalixir/gigalixir-buildpack-distillery.git
  - To run mix release or mix distillery.release
  - This is only run if you have a rel/config.exs file present.
- https://github.com/gigalixir/gigalixir-buildpack-releases

  - To run mix release if you are running Elixir 1.9 and using the built-in releases
  - This is only run if you have a config/releases.exs file present.
- https://github.com/gigalixir/gigalixir-buildpack-mix.git
  - To set up your Procfile correctly
  - This is only run if you *don't* have a rel/config.exs file present.

We only build the master branch and ignore other branches. When building, we cache compiled files and dependencies so you do not have to repeat the work on every deploy. We support git submodules.

Once your slug is built, we upload it to slug storage and we combine it with a config to create a new release for your app. The release is tagged with a `version` number which you can use later on if you need to rollback to this release.

Then we create or update your Kubernetes configuration to deploy the app. We create a separate Kubernetes namespace for every app, a service account, an ingress for HTTP traffic, an ingress for SSH traffic, a TLS certificate, a service, and finally a deployment which creates pods and containers.

The container that runs your app is a derivative of heroku/cedar:14. The entrypoint is a script that sets up necessary environment variables including those from your *app configuration*. It also starts an SSH server, installs your SSH keys, downloads the current slug, and executes it. We automatically generate and set up your erlang cookie, distributed node name, and Phoenix secret key base for you. We also set up the Kubernetes permissions and libcluster selector you need to *cluster your nodes*. We poll for your SSH keys every minute in case they have changed.

At this point, your app is running. The Kubernetes ingress controller is routing traffic from your host to the appropriate pods and terminating SSL/TLS for you automatically. For more information about how SSL/TLS works, see *How SSL/TLS Works*.

If at any point, the deploy fails, we rollback to the last known good release.

To see how we do zero-downtime deploys, see *How to Set Up Zero-Downtime Deploys*.

7.4 How SSL/TLS Works

We use kube-lego for automatic TLS certificate generation with Let's Encrypt. For more information, see kube-lego's documentation. When you add a custom domain, we create a Kubernetes ingress for you to route traffic to your app. kube-lego picks this up, obtains certificates for you and installs them. Our ingress controller then handles terminating SSL traffic before sending it to your app.

7.5 Life of a Hot Upgrade

There is an extra flag you can pass to deploy by hot upgrade instead of a restart. You have to make sure you bump your app version in your `mix.exs`. Distillery autogenerates your appup file, but you can supply a custom appup file if you need it. For more information, look at the Distillery appup documentation.

```
git -c http.extraheader="GIGALIXIR-HOT: true" push gigalixir master
```

A hot upgrade follows the same steps as a regular deploy, except for a few differences. In order for distillery to build an upgrade, it needs access to your old app so we download it and make it available in the build container.

Once the slug is generated and uploaded, we execute an upgrade script on each run container instead of restarting. The upgrade script downloads the new slug, and calls Distillery's upgrade command. Your app should now be upgraded in place without any downtime, dropped connections, or loss of in-memory state.

CHAPTER 8

How to clean your build cache

There is an extra flag you can pass to clean your cache before building in case you need it, but you need git 2.9.0 or higher for it to work. For information on how to install the latest version of git on Ubuntu, see this stackoverflow question.

```
git -c http.extraheader="GIGALIXIR-CLEAN: true" push gigalixir master
```

CHAPTER 9

Known Issues

- Warning: Multiple default buildpacks reported the ability to handle this app. The first buildpack in the list below will be used.
  - This warning is safe to ignore. It is a temporary warning due to a workaround.
- curl: (56) GnuTLS recv error (-110): The TLS connection was non-properly terminated.
  - Currently, the load balancer for domains under gigalixirapp.com has a request timeout of 30 seconds. If your request takes longer than 30 seconds to respond, the load balancer cuts the connection. Often, the cryptic error message you will see when using curl is the above. The load balancer for custom domains does not have this problem.
- php apps don't work well with the default stack, gigalixir-18. If you are deploying php, please downgrade your stack to giglaixir-16 with something like `gigalixir stack:set -s gigalixir-16`. Th e reason is because gigalixir-18 is based on heroku-18 which does not have libreadline.so preinstalled for some reason where giglaixir-16, based on heroku-16, does.
- Did not find exactly 1 release
  - This can happen for a few different reasons, but usually clearing your build cache or retrying will resolve it. In some cases, if you added `release=true` to your `elixir_buildpack.config` file, it caches the release and is never deleted even when you bump the app version in your mix.exs. This results in two release folders and gigalixir does not know which release you intend to deploy and errors out. Clearing the cache resolves this issue. In some cases, if two deploys are running concurrently, you can end up with two release tarballs at the same time. This is a known issue we intend to fix, but usually re-running the deploy will work fine since it is a race condition.

CHAPTER 10

Can I run my app in AWS instead of Google Cloud Platform? What about Europe?

Yes, if your current infrastructure is on AWS, you'll probably want to run your gigalixir app on AWS too. Or if most of your users are in Europe, you probably want to host your app in Europe. We currently support GCP v2018-us-central1 and GCP europe-west1 as well as AWS us-east-1 and AWS us-west-2. When creating your app with `gigalixir create` simply specify the `--cloud=aws` and `--region=us-east-1` options.

Once the app is created, it's difficult to migrate to another region. If you want to do this, Heroku's guide is a good overview of what you should consider. If you don't mind downtime, the transition could be easy, but unfortunately gigalixir isn't able to do it for you with a button press. See https://devcenter.heroku.com/articles/app-migration

One thing to keep in mind is that Gigalixir Postgres databases are as of right now only available in GCP/v2018-us-central1 and GCP/europe-west1, however, we can set up a database for you in AWS manually if you like. Just *contact us* and we'll create one for you. We plan to add AWS to the Gigalixir CLI soon.

If you don't see the region you want, please *contact us* and let us know. We open new regions based purely on demand.

CHAPTER 11

Can I use a custom Procfile?

Definitely! If you are using mix (not distillery) and you have a `Procfile` at the root of your repo, we'll use it instead of the default one. If you are using Distillery, you'll have to use distillery overlays to include the Procfile inside your release tarball i.e. slug.

The only gotcha is that if you want remote console to work, you'll want to make sure the node name and cookie are set properly. For example, your `Procfile` should look something like this.

```
web: elixir --name $MY_NODE_NAME --cookie $MY_COOKIE -S mix phoenix.server
```

CHAPTER 12

How do I specify my Elixir, Erlang, Node, NPM, etc versions?

Your Elixir and Erlang versions are handled by the heroku-buildpack-elixir buildpack. To configure, see the heroku-buildpack-elixir configuration. In short, you specify them in a `elixir_buildpack.config` file.

Node and NPM versions are handled by the heroku-buildpack-phoenix-static buildpack. To configure, see the heroku-buildpack-phoenix-static configuration. In short, you specify them in a `phoenix_static_buildpack.config` file.

Supported Elixir and erlang versions can be found at https://github.com/HashNuke/heroku-buildpack-elixir#version-support

CHAPTER 13

How do I specify which buildpacks I want to use?

Normally, the buildpack you need is auto-detected for you, but in some cases, you may want to specify which buildpacks you want to use. To do this, create a `.buildpacks` file and list each buildpack you want to use. For example, the default buildpacks for Elixir apps using distillery would look like this

```
https://github.com/HashNuke/heroku-buildpack-elixir
https://github.com/gjaldon/heroku-buildpack-phoenix-static
https://github.com/gigalixir/gigalixir-buildpack-distillery.git
```

The default buildpacks for Elixir apps running mix looks like this

```
https://github.com/HashNuke/heroku-buildpack-elixir
https://github.com/gjaldon/heroku-buildpack-phoenix-static
https://github.com/gigalixir/gigalixir-buildpack-mix.git
```

Note the last buildpack. It's there to make sure your `Procfile` is set up correctly to run on gigalixir. It basically makes sure you have your node name and cookie set correctly so that remote console, migrate, observer, etc will work.

CHAPTER 14

How do I deploy an umbrella app?

Umbrella apps are deployed the same way, but the buildpacks need to know which internal app is your Phoenix app. Set your `phoenix_relative_path` in your `phoenix_static_buildpack.config` file, see the heroku-buildpack-phoenix-static configuration for more details.

When running migrations, we need to know which internal app contains your migrations. Use the `--migration_app_name` flag on `gigalixir ps:migrate`.

If you have multiple Distillery releases in your `rel/config.exs` file, be sure to set your default release to the one you want to deploy. See *How to specify which release, environment, or profile to build*.

If you have multiple Phoenix apps in the umbrella, you'll need to use something like this master_proxy to proxy requests to the two apps.

CHAPTER 15

Can I deploy an app that isn't at the root of my repository?

If you just want to push a subtree, try

```
git subtree push --prefix my-sub-folder gigalixir master
```

If you want to push the entire repo, but run the app from a subfolder, it becomes a bit trickier, but this pull request should help you. https://github.com/jesseshieh/nonroot/pull/1/files

CHAPTER 16

How to do blue-green or canary deploys?

This feature is in beta as of 3/19/2019. You'll need the CLI v1.0.19 or later.

Apps on Gigalixir can be assigned another app as its canary. An arbitrary weight can also be assigned to control the traffic between the two apps. For example, if you have `my-app` with a canary assigned to it called `my-app-canary` with weight of 10, then `my-app` will receive 90% of the traffic and `my-app-canary` will receive 10% of the traffic. If you want to do blue-green deploys, simply flip the traffic between 0 and 100 to control which app receives the traffic. For example,

```
# create the "blue" app
gigalixir create --name my-app-blue
git remote rename gigalixir blue

# create the "green" app
gigalixir create --name my-app-green
git remote rename gigalixir green

# deploy the app to blue
git push blue master

# wait a few minutes and ensure the app is running
curl https://my-app-blue.gigalixirapp.com/

# deploy the app to green
git push green master

# wait a few minutes to ensure the app is running
curl https://my-app-green.gigalixirapp.com/

# watch the logs on both apps
gigalixir logs -a my-app-blue
gigalixir logs -a my-app-green

# set the canary, this should have no effect because the weight is 0
gigalixir canary:set -a my-app-blue -c my-app-green -w 0
```

(continues on next page)

(continued from previous page)

```
# increase the weight to the canary
gigalixir canary:set -a my-app-blue -w 10

# ensure traffic is split as expected by watching the logs
# flip traffic completely to green
gigalixir canary:set -a my-app-blue -w 100

# ensure traffic is going totally to green by watching the logs
# to delete a canary, run
gigalixir canary:unset -a my-app-blue -c my-app-green
```

Notice that with canaries, only the domain for `my-app-blue` gets redirected. Traffic to my-app-green.gigalixirapp.com goes entirely to `my-app-green`.

If you have custom domains defined on `my-app-blue`, traffic to those will also be shaped by the canary, but custom domains on `my-app-green` will still go entirely to `my-app-green`.

CHAPTER 17

Frequently Asked Questions

17.1 *What versions of Phoenix do you support?*

All versions.

17.2 *What versions of Elixir and OTP do you support?*

All versions of Elixir and OTP. See *How do I specify my Elixir, Erlang, Node, NPM, etc versions?*. Some buildpacks don't have the bleeding edge versions so those might not work, but they will eventually.

Can I have multiple custom domains?

Yes! Just follow *How to Set Up a Custom Domain* for each domain.

17.3 *Do you support non-Elixir apps?*

Yes, we support any language that has a buildpack, but hot upgrades, remote observer, etc probably won't work. Built-in buildpacks include

- multi
- ruby
- nodejs
- clojure
- python
- java
- gradle
- scala

- play
- php
- go
- erlang
- static

For details, see https://github.com/gliderlabs/herokuish/tree/v0.3.36/buildpacks

If the buildpack you need is not built-in, you can specify the buildpack(s) you want by listing them in a `.buildpacks` file.

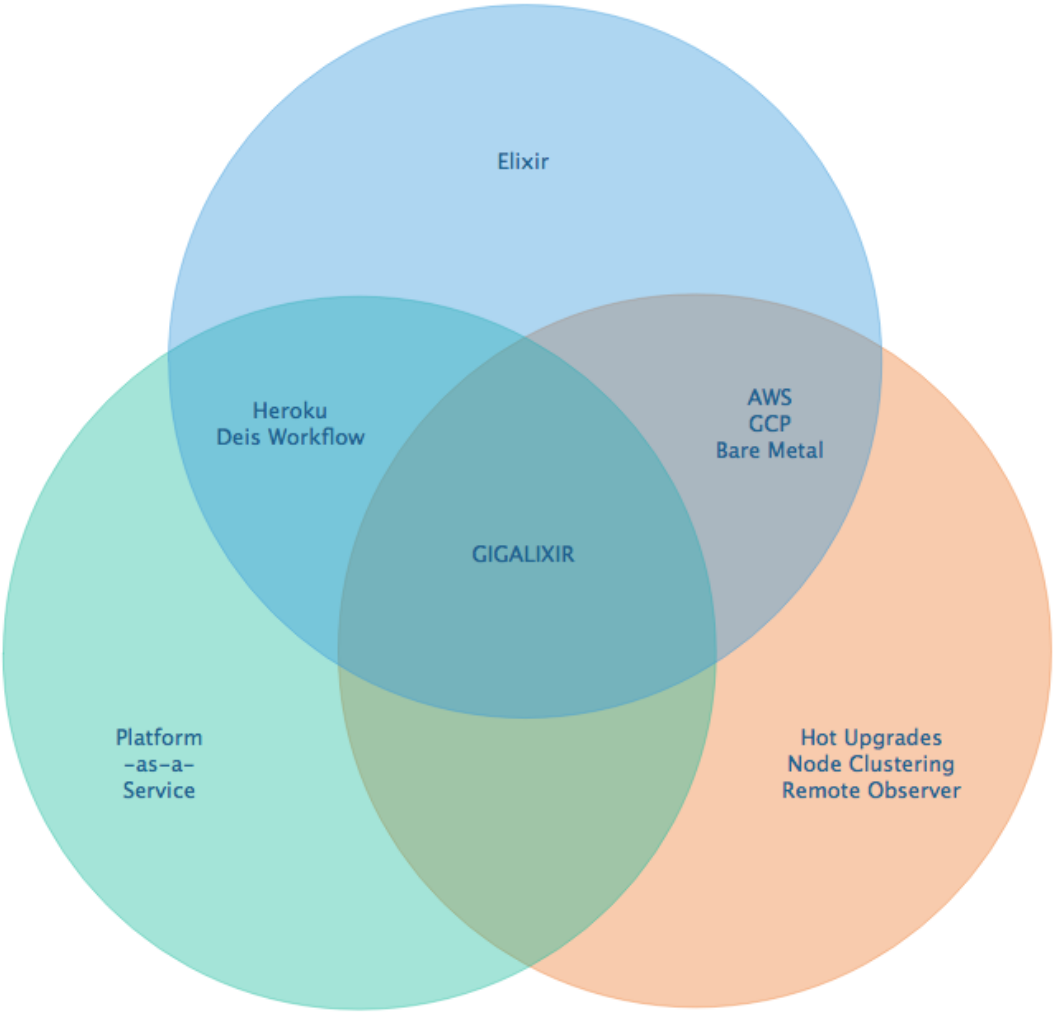
For an example, see *How to deploy a Ruby app*.

17.4 What is Elixir? What is Phoenix?

This is probably best answered by taking a look at the elixir homepage and the phoenix homepage.

17.5 How is Gigalixir different from Heroku and Deis Workflow?

For a feature comparison table between Gigalixir and Heroku see, *Gigalixir vs Heroku Feature Comparison*.



Heroku is a really great platform and much of Gigalixir was designed based on their excellent twelve-factor methodology. Heroku and Gigalixir are similar in that they both try to make deployment and operations as simple as possible. Elixir applications, however, aren't very much like most other apps today written in Ruby, Python, Java, etc. Elixir apps are distributed, highly-available, hot-upgradeable, and often use lots of concurrent long-lived connections. Gigalixir made many fundamental design choices that ensure all these things are possible.

For example, Heroku restarts your app every 24 hours regardless of if it is healthy or not. Elixir apps are designed to be long-lived and many use in-memory state so restarting every 24 hours sort of kills that. Heroku also limits the number of concurrent connections you can have. It also has limits to how long these connections can live. Heroku isolates each instance of your app so they cannot communicate with each other, which prevents node clustering. Heroku also restricts SSH access to your containers which makes it impossible to do hot upgrades, remote consoles, remote observers, production tracing, and a bunch of other things. The list goes on, but suffice it to say, running an Elixir app on Heroku forces you to give up a lot of the features that drew you to Elixir in the first place.

Deis Workflow is also really great platform and is very similar to Heroku, except you run it your own infrastructure. Because Deis is open source and runs on Kubernetes, you *could* make modifications to support node clustering and remote observer, but they won't work out of the box and hot upgrades would require some fundamental changes to the way Deis was designed to work. Even so, you'd still have to spend a lot of time solving problems that Gigalixir has already figured out for you.

On the other hand, Heroku and Deis are more mature products that have been around much longer. They have more

features, but we are working hard to fill in the holes. Heroku and Deis also support languages other than Elixir.

17.6 *I thought you weren't supposed to SSH into docker containers!?*

There are a lot of reasons not to SSH into your docker containers, but it is a tradeoff that doesn't fit that well with Elixir apps. We need to allow SSH in order to connect a remote observer to a production node, drop into a remote console, and do hot upgrades. If you don't need any of these features, then you probably don't need and probably shouldn't SSH into your containers, but it is available should you want to. Just keep in mind that full SSH access to your containers means you have almost complete freedom to do whatever you want including shoot yourself in the foot. Any manual changes you make during an SSH session will also be wiped out if the container restarts itself so use SSH with care.

17.7 *Why do you download the slug on startup instead of including the slug in the Docker image?*

Great question! The short answer is that after a hot-upgrade, if the container restarts, you end up reverting back to the slug included in the container. By downloading the slug on startup, we can always be sure to pull the most current slug even after a hot upgrade.

This sort of flies in the face of a lot of advice about how to use Docker, but it is a tradeoff we felt was necessary in order to support hot upgrades in a containerized environment. The non-immutability of the containers can cause problems, but over time we've ironed them out and feel that there is no longer much downside to this approach. All the headaches that came as a result of this decision are our responsibility to address and shouldn't affect you as a customer. In other words, you reap the benefits while we pay the cost, which is one of the ways we provide value.

17.8 *How do I add worker processes?*

Heroku and others allow you to specify different types of processes under a single app such as workers that pull work from a queue. With Elixir, that is rarely needed since you can spawn asynchronous tasks within your application itself. Elixir and OTP provide all the tools you need to do this type of stuff among others. For more information, see Background Jobs in Phoenix which is an excellent blog post. If you really need to run an Redis-backed queue to process jobs, take a look at Exq, but consider whether you really need Exq.

17.9 *What if Gigalixir shuts down?*

Gigalixir is running profitably and has plenty of funding. There is no reason to think Gigalixir will shut down.

17.10 *My git push was rejected*

Try force pushing with

```
git push -f gigalixir master
```

CHAPTER 18

Clustering Nodes

First of all, be sure you are using Distillery and not mix for your deploys. Clustering won't work with just mix. For instructions on using distillery, see *Mix vs Distillery vs Elixir Releases*.

We use libcluster to manage node clustering. For more information, see libcluster's documentation.

To install libcluster, add this to the deps list in `mix.exs`

```
{:libcluster, "~> 2.0.3"}
```

If you are on Elixir 1.3 or lower, add `libcluster` and `:ssl` to your applications list. Elixir 1.4 and up detect your applications list for you.

If you are running erlang/OTP 21 or higher, you need to use libcluster 3.0 or higher and add the following to your `application.ex` file.

```
# libcluster 3.0+ only
children = [
  {Cluster.Supervisor, [Application.get_env(:libcluster, :topologies), [name:␣
↪GigalixirGettingStarted.ClusterSupervisor]]},
  ... # other children
]
```

Your app configuration needs to have something like this in it. For a full example, see gigalixir-getting-started's prod.exs file.

```
...
config :libcluster,
  topologies: [
    k8s_example: [
      strategy: Cluster.Strategy.Kubernetes,
      config: [
        # For Elixir Releases, use System.get_env instead of the distillery env vars␣
↪below.
        kubernetes_selector: "${LIBCLUSTER_KUBERNETES_SELECTOR}",
```

(continues on next page)

(continued from previous page)

```
        kubernetes_node_basename: "${LIBCLUSTER_KUBERNETES_NODE_BASENAME}"]]]
...
```

Gigalixir handles permissions so that you have access to Kubernetes endpoints and we automatically set your node name and erlang cookie so that your nodes can reach each other. We don't firewall each container from each other like Heroku does. We also automatically set the environment variables `LIBCLUSTER_KUBERNETES_SELECTOR`, `LIBCLUSTER_KUBERNETES_NODE_BASENAME`, `APP_NAME`, and `MY_POD_IP` for you. See gigalixir-run for more details.

CHAPTER 19

How to use a custom vm.args

Gigalixir generates a default `vm.args` file for you and tells Distillery to use it by setting the `VMARGS_PATH` environment variable. By default, it is set to `/release-config/vm.args`. If you want to use a custom `vm.args`, we recommend you follow these instructions.

Disable Gigalixir's default vm.args

```
gigalixir config:set GIGALIXIR_DEFAULT_VMARGS=false
```

Create a `rel/vm.args` file in your repository. It might look something like gigalixir-getting-started's vm.args file.

Lastly, you need to modify your distillery config so it knows where to find your `vm.args` file. Something like this. For a full example, see gigalixir-getting-started's rel/config.exs file.

```
...
environment :prod do
  ...
  # this is just to get rid of the warning. see https://github.com/bitwalker/
↪distillery/issues/140
  set cookie: :"${MY_COOKIE}"
  set vm_args: "rel/vm.args"
end
...
```

After a new deploy, verify by SSH'ing into your instance and inspecting your release's vm.arg file like this

```
gigalixir ps:ssh
cat /app/var/vm.args
```

CHAPTER 20

Tiers

Gigalixir offers 2 tiers of pricing. The free tier is free, but you are limited to 1 instance up to size 0.5 and 1 free tier database. Also, on the free tier, if you haven't deployed anything for over 30 days, we will send you an email to remind you to keep your account active. If you do not, your app may be scaled down to 0 replicas. We know this isn't ideal, but we think it is better than sleeping instances and we appreciate your understanding since the free tier does cost a lot to run.

Instance Feature	FREE Tier	STANDARD Tier
Zero-downtime deploys	YES	YES
Websockets	YES	YES
Automatic TLS	YES	YES
Log Aggregation	YES	YES
Log Tailing	YES	YES
Hot Upgrades	YES	YES
Remote Observer	YES	YES
No Connection Limits	YES	YES
No Daily Restarts	YES	YES
Custom Domains	YES	YES
Postgres-as-a-Service	YES	YES
SSH Access	YES	YES
Vertical Scaling		YES
Horizontal Scaling		YES
Clustering		YES
Multiple Apps		YES
Team Permissions		YES
No Inactivity Checks		YES

Database Feature	FREE Tier	STANDARD Tier
SSL Connections	YES	YES
Data Import/Export	YES	YES
Data Encryption		YES
Dedicated CPU		YES*
Dedicated Memory		YES
Dedicated Disk		YES
No Connection Limits		YES*
No Row Limits		YES
Backups		YES
Scalable/Upgradeable		YES
Automatic Data Migration		YES
Postgres Extensions		YES
Role Management		YES

- Only size 4 and above have dedicated CPU. See *Database Sizes & Pricing*.
- Databases still have connection limits based on Google Cloud SQL limits. See https://cloud.google.com/sql/docs/postgres/quotas#fixed-limits

CHAPTER 21

Gigalixir vs Heroku Feature Comparison

Feature	Gigalixir FREE Tier	Gigalixir STANDARD Tier	Heroku Free	Heroku Standard	Heroku Performance
Websockets	YES	YES	YES	YES	YES
Log Aggregation	YES	YES	YES	YES	YES
Log Tailing	YES	YES	YES	YES	YES
Custom Domains	YES	YES	YES	YES	YES
Postgres-as-a-Service	YES	YES	YES	YES	YES
No sleeping	YES	YES		YES	YES
Automatic TLS	YES	YES		YES	YES
Preboot	YES	YES		YES	YES
Zero-downtime deploys	YES	YES			
SSH Access	YES	YES			
Hot Upgrades	YES	YES			
Remote Observer	YES	YES			
No Connection Limits	YES	YES			
No Daily Restarts	YES	YES			
Flexible Instance Sizes		YES			
Clustering		YES			
Horizontal Scaling		YES		YES	YES
Built-in Metrics				YES	YES
Threshold Alerts				YES	YES
Dedicated Instances					YES
Autoscaling					YES

CHAPTER 22

Pricing Details

In the free tier, everything is no-credit-card free. Once you upgrade to the standard tier, you pay $10 for every 200MB of memory per month. CPU, bandwidth, and power are free.

See our cost estimator to calculate how much you should expect to pay each month. Keep reading for exactly how we compute your bill.

Every month after you sign up on the same day of the month, we calculate the number of replica-size-seconds used, multiply that by $0.00001866786, and charge your credit card.

replica-size-seconds is how many replicas you ran multiplied by the size of each replica multiplied by how many seconds they were run. This is aggregated across all your apps and is prorated to the second.

For example, if you ran a single 0.5 size replica for 31 days, you will have used

```
(1 replica) * (0.5 size) * (31 days) = 1339200 replica-size-seconds.
```

Your monthly bill will be

```
1339200 * $0.00001866786 = $25.00.
```

If you ran a 1.0 size replica for 10 days, then scaled it up to 3 replicas, then 10 days later scaled the size up to 2.0 and it was a 30-day month, then your usage would be

```
(1 replica) * (1.0 size) * (10 days) + (3 replicas) * (1.0 size) * (10 days) + (3␣
↪replicas) * (2.0 size) * (10 days) = 8640000 replica-size-seconds
```

Your monthly bill will be

```
8640000 * $0.00001866786 = $161.29.
```

For database pricing, see *Database Sizes & Pricing*.

CHAPTER 23

Database Sizes & Pricing

In the free tier, the database is free, but it is really not suitable for production use. It is a multi-tenant postgres database cluster with shared CPU, memory, and disk. You are limited to 2 connections, 10,000 rows, and no backups. Idle connections are terminated after 5 minutes. If you want to upgrade your database, you'll have to migrate the data yourself. For a complete feature comparison see *Tiers*.

In the standard tier, database sizes are defined as a single number for simplicity. The number defines how many GBs of memory your database will have. Supported sizes include 0.6, 1.7, 4, 8, 16, 32, 64, and 128. Sizes 0.6 and 1.7 share CPU with other databases. All other sizes have dedicated CPU, 1 CPU for every 4 GB of memory. For example, size 4 has 1 dedicated CPU and size 64 has 16 dedicated CPUs. All databases start with 10 GB disk and increase automatically as needed.

Size	Price / Month	RAM	Rollback Days	Connection Limit	Storage Limit
0.6	$25	0.6 GB	7	25	25 GB
1.7	$50	1.7 GB	7	50	50 GB
4	$100	4 GB	7	100	100 GB
8	$200	8 GB	7	200	200 GB
16	$400	16 GB	7	250	400 GB
32	$800	32 GB	7	300	800 GB
64	$1600	64 GB	7	400	1.6 TB
128	$3200	128 GB	7	500	3.2 TB

Prices are prorated to the second.

For more, see *How to provision a Standard PostgreSQL database* and *How to provision a Free PostgreSQL database*.

CHAPTER 24

Replica Sizing

- A replica is a docker container that your app runs in.
- Replica sizes are available in increments of 0.1 between 0.2 and 384, but for the higher sizes you'll need to *contact us* first.
- 1 size unit is 1GB memory and 1 CPU share.
- 1 CPU share is 200m as defined using Kubernetes CPU requests or roughly 20% of a core guaranteed.
  - If you are on a machine with other containers that don't use much CPU, you can use as much CPU as you like.
- Memory is defined using Kuberenetes memory requests.
  - If you are on a machine with other machines that don't use much memory, you can use as much memory as you like.
- Memory and CPU sizes can not be adjusted separately.

CHAPTER 25

Releases

One common pitfall for beginners is how releases differ from running apps with Mix. In development, you typically have access to Mix tasks to run your app, migrate your database, etc. In production, we use releases. With releases, your code is distributed in it's compiled form and is almost no different from an Erlang release. You no longer have access to Mix commands. However, in return, you also have access to hot upgrades and smaller slug sizes, and a "single package which can be deployed anywhere, independently of an Erlang/Elixir installation. No dependencies, no hassle" [1].

[1]: https://github.com/bitwalker/distillery

CHAPTER 26

Limits

Gigalixir is designed for Elixir/Phoenix apps and it is common for Elixir/Phoenix apps to have many connections open at a time and to have connections open for long periods of time. Because of this, we do not limit the number of concurrent connections or the duration of each connection[1][2].

We also know that Elixir/Phoenix apps are designed to be long-lived and potentially store state in-memory so we do not restart replicas arbitrarily. In fact, replicas should not restart at all, unless there is an extenuating circumstance that requires it. For apps that require extreme high availability, we suggest that your app be able to handle node restarts just as you would for any app not running on Gigalixir.

[1] Because Gigalixir runs on Google Compute Engine, you may bump into an issue with connections that stay idle for 10m. For more information and how to work around it, see https://cloud.google.com/compute/docs/troubleshooting
[2] We do have a timeout of 60 minutes for connections after an nginx configuration reload. If you have a long-lived websocket connection and our nginx configuration is reloaded, the connection will be dropped 60 minutes later. Unfortunately, nginx reloads happen frequently under Kubernetes.

CHAPTER 27

Monitoring

Gigalixir doesn't provide any monitoring out of the box, but we are working on it. Also, you can always use a remote observer to inspect your node. See, *How to Launch a Remote Observer*.

CHAPTER 28

Using Environment Variables in your App

Environment variables with Elixir, Distillery, and releases in general are one of those things that always trip up beginners. I think Distillery's Runtime Configuration explains it better than I can, but in short, never use `System.get_env("FOO")` in your `prod.exs`. Always use `"${FOO}"` instead.

Gigalixir automatically sets `REPLACE_OS_VARS=true` for you so all you have to do to introduce a new `MY_CONFIG` env var is add something like this to your `config.exs` file

```
...
config :myapp,
    my_config: "${MY_CONFIG}"
...
```

Then set the `MY_CONFIG` environment variable, by running

```
gigalixir config:set MY_CONFIG=foo
```

In your app code, access the environment variable using

```
Application.get_env(:myapp, :my_config) == "foo"
```

CHAPTER 29

Troubleshooting

If your app isn't working and you're seeing either 504s or an "unhealthy" message, you're in the right place. The first places to check for clues are *gigalixir logs* and *gigalixir ps*. If nothing pops out at you there, keep reading.

A 504 means that our load balancer isn't able to reach your app. This is usually because the app isn't running. An app that isn't running is usually failing health checks and we constantly restart apps that fail health checks in hopes that it will become healthy.

If you've just deployed, and you're not seeing 504s, but you're still seeing the old version of your app instead of the new version, it's the same problem. This happens when the new version does not pass health checks. When the new version doesn't pass health checks, we don't route traffic to it and we don't terminate the old version.

Our health checks simply check that your app is listening on port $PORT. If you're running a non-HTTP Elixir app, but need to just get health checks to pass, take a look at https://github.com/jesseshieh/elixir-tcp-accept-and-close

If you're using Mix, see *troubleshooting mix*.

If you're using Distillery, see *troubleshooting distillery*.

If you're using Elixir Releases, see *troubleshooting Elixir releases*.

29.1 Mix

Let's verify that your app works locally.

Run the following commands

```
mix deps.get
mix compile
SECRET_KEY_BASE="$(mix phx.gen.secret)" MIX_ENV=prod DATABASE_URL="postgresql://
↪user:pass@localhost:5432/foo" PORT=4000 mix phx.server
curl localhost:4000
```

If it doesn't work, the first thing to check is your `prod.exs` file. Often, it is missing an `http` configuration or there is a typo in the `FooWeb.Endpoint` module name.

If everything works locally, you might be running a different version of Elixir in production. See *How do I specify my Elixir, Erlang, Node, NPM, etc versions?*.

Another possibility is that your app is running out of memory and can't start up properly. To fix this, try scaling up. See *How to Scale an App*.

If the above commands still do not succeed and your app is open source, then please *contact us for help*. If not open source, *contact us* anyway and we'll do our best to help you.

29.2 Distillery

If you're having trouble getting things working, you can verify a few things locally.

First, try generating and running a Distillery release locally by running

```
mix deps.get
mix compile
export SECRET_KEY_BASE="$(mix phx.gen.secret)"
export DATABASE_URL="postgresql://user:pass@localhost:5432/foo"
MIX_ENV=prod mix distillery.release --env=prod
# if you are a running distillery below 2.1, then run this instead: MIX_ENV=prod mix␣
↪release --env=prod
APP_NAME=gigalixir_getting_started
MY_HOSTNAME=example.com MY_COOKIE=secret REPLACE_OS_VARS=true MY_NODE_NAME=foo@127.0.
↪0.1 PORT=4000 _build/prod/rel/$APP_NAME/bin/$APP_NAME foreground
curl localhost:4000
```

Don't forget to replace `gigalixir_getting_started` with your own app name. Also, change/add the environment variables as needed.

You can safely ignore Kubernetes errors like `[libcluster:k8s_example]` errors because you probably aren't running inside Kubernetes.

If they don't work, the first place to check is `prod.exs`. Make sure you have `server: true` somewhere and there are no typos.

In case static assets don't show up, you can try the following and then re-run the commands above.

```
cd assets
npm install
npm run deploy
cd ..
mix phx.digest
```

If your problem is with one of the buildpacks, try running the full build using Docker and Herokuish by running

```
APP_ROOT=$(pwd)
rm -rf /tmp/gigalixir/cache
rm -rf _build
mkdir -p /tmp/gigalixir/cache
docker run -it --rm -v $APP_ROOT:/tmp/app -v /tmp/gigalixir/cache/:/tmp/cache us.gcr.
↪io/gigalixir-152404/herokuish
```

Or to inspect closer, run

```
docker run -it --rm -v $APP_ROOT:/tmp/app -v /tmp/gigalixir/cache/:/tmp/cache --
↪entrypoint=/bin/bash us.gcr.io/gigalixir-152404/herokuish

# and then inside the container run
build-slug

# inspect /app folder
# check /tmp/cache
```

If everything works locally, you might be running a different version of Elixir in production. See *How do I specify my Elixir, Erlang, Node, NPM, etc versions?*.

Another possibility is that your app is running out of memory and can't start up properly. To fix this, try scaling up. See *How to Scale an App*.

If the above commands still do not succeed and your app is open source, then please *contact us for help*. If not open source, *contact us* anyway and we'll do our best to help you.

29.3 Elixir Releases

If you're having trouble getting things working, you can verify a few things locally.

First, try generating and running a release locally by running

```
mix deps.get
SECRET_KEY_BASE="$(mix phx.gen.secret)"
DATABASE_URL="postgresql://user:pass@localhost:5432/foo"
MIX_ENV=prod mix release
APP_NAME=gigalixir_getting_started
PORT=4000 _build/prod/rel/$APP_NAME/bin/$APP_NAME start
curl localhost:4000
```

Don't forget to replace `gigalixir_getting_started` with your own app name. Also, change/add the environment variables as needed.

You can safely ignore Kubernetes errors like `[libcluster:k8s_example]` errors because you probably aren't running inside Kubernetes.

If they don't work, the first place to check is `prod.exs`. Make sure you have `server: true` somewhere and there are no typos.

In case static assets don't show up, you can try the following and then re-run the commands above.

```
cd assets
npm install
npm run deploy
cd ..
mix phx.digest
```

If your problem is with one of the buildpacks, try running the full build using Docker and Herokuish by running

```
APP_ROOT=$(pwd)
rm -rf /tmp/gigalixir/cache
rm -rf _build
mkdir -p /tmp/gigalixir/cache
docker run -it --rm -v $APP_ROOT:/tmp/app -v /tmp/gigalixir/cache/:/tmp/cache us.gcr.
↪io/gigalixir-152404/herokuish
```

Or to inspect closer, run

```
docker run -it --rm -v $APP_ROOT:/tmp/app -v /tmp/gigalixir/cache/:/tmp/cache --
↪entrypoint=/bin/bash us.gcr.io/gigalixir-152404/herokuish

# and then inside the container run
build-slug

# inspect /app folder
# check /tmp/cache
```

If everything works locally, you might be running a different version of Elixir in production. See *How do I specify my Elixir, Erlang, Node, NPM, etc versions?*.

Another possibility is that your app is running out of memory and can't start up properly. To fix this, try scaling up. See *How to Scale an App*.

If the above commands still do not succeed and your app is open source, then please *contact us for help*. If not open source, *contact us* anyway and we'll do our best to help you.

29.4 Common Errors

A good first thing to try when you get a *git push* error is *cleaning your build cache*.

- My deploy succeeded, but nothing happened.
  - When `git push gigalixir master` succeeds, it means your code was compiled and built without any problems, but there can still be problems during runtime. Other platforms will just let your app fail, but gigalixir performs tcp health checks on port 4000 on your new release before terminating the old release. So if your new release is failing health checks, it can appear as if nothing is happening because in a sense, nothing is. Check `gigalixir logs` for any startup errors.
- My app takes a long time to startup.
  - Most likely, this is because your CPU reservation isn't enough and there isn't any extra CPU available on the machine to give you. Try scaling up your instance sizes. See *How to Scale an App*.
- failed to connect: ** (Postgrex.Error) FATAL 53300 (too_many_connections): too many connections for database
  - If you have a free tier database, the number of connections is limited. Try lowering the `pool_size` in your `prod.exs` to 2, or if you're using `prod.secret.exs` setting the `POOL_SIZE` environment variable using `gigalixir config:set POOL_SIZE=2`.
- ~/.netrc access too permissive: access permissions must restrict access to only the owner
  - run `chmod og-rwx ~/.netrc`
- `git push gigalixir master` asks for my password
  - First try running `gigalixir login` and try again. If that doesn't work, try resetting your git remote by running `gigalixir git:remote $APP` and trying again.
- remote: cp: cannot overwrite directory '/tmp/cache/node_modules/phoenix_html' with non-directory
  - Try *cleaning your build cache*. Looks like something changed in your app that makes the cache non-overwritable.
- `conn.remote_ip` has `127.0.0.1` instead of the real client ip

  - Try using https://github.com/kbrw/plug_forwarded_peer or otherwise use the `X-Forwarded-For` header instead. Gigalixir apps run behind load balancers which write the real client ip in that header.

- (File.Error) could not read file "foo/bar": no such file or directory
  - Often, this means that Distillery did not package the `foo` directory into your release tarball. Try using Distillery Overlays to add the `foo` directory. For example, adjusting your `rel/config.exs` to something like this

```
release :gigalixir_getting_started do
  set version: current_version(:gigalixir_getting_started)
  set applications: [
    :runtime_tools
  ]
  set overlays: [
    {:copy, "foo", "foo"}
  ]
end
```

    For more, see https://github.com/bitwalker/distillery/blob/master/docs/Overlays.md

- cd: /tmp/build/./assets: No such file or directory
  - This means the Phoenix static buildpack could not find your assets folder. Either specify where it is or remove the buildpack. To specify, configure the buildpack following https://github.com/gjaldon/heroku-buildpack-phoenix-static. To remove, create a `.buildpacks` file with the buildpacks you need. For example, just `https://github.com/HashNuke/heroku-buildpack-elixir`

- SMTP/Email Network Failures e.g. {:network_failure, 'smtp.mailgun.org', {:error, :timeout}}
  - Google Cloud Engine does not allow certain email ports like 587. See https://cloud.google.com/compute/docs/tutorials/sending-mail/ Try using port 2525. See https://cloud.google.com/compute/docs/tutorials/sending-mail/using-mailgun

- unknown command: MIX_ENV=prod mix phx.server
  - If you are you are using a custom Procfile with an environment variable at the front of the command, you'll get this error. Try adding `env` to the front of the command. See https://github.com/ddollar/foreman/issues/265. We use the most command Ruby Foreman which behaves differently from Heroku's for this situation.

- init terminating in do_boot ({cannot get bootfile,no_dot_erlang.boot})
  - This is an issue described here: https://github.com/bitwalker/distillery/issues/426 Try either upgrading Distillery to 1.5.3 or downgrading OTP below 21.

- Could not invoke task "release": –env : Unknown option
  - This happens when you upgrade to elixir 1.9, but are still using distillery older than 2.1. Upgrade distillery to fix this issue, but be sure to also change your rel/config.exs file. Mix.Releases.Config needs to be renamed to Distillery.Releases.Config

- sh: 1: mix: not found
  - If you have an old Phoenix project where a `package.json` file exists in the project root folder, the `herokuish` buildpack might mistakenly recognize it as a Node.js project, and thus fail to build it properly. You may need to manually add a `.buildpacks` file in your root folder, as documented in the "Specify Buildpacks" sections above.

CHAPTER 30

Support/Help

Feel free to email help@gigalixir.com for any questions or issues, we generally respond quickly.

CHAPTER 31

The Gigalixir Command-Line Interface

The Gigalixir Command-Line Interface or CLI is a tool you install on your local machine to control Gigalixir.

31.1 How to Install the CLI

See *Install the Command-Line Interface*.

31.2 How to Upgrade the CLI

To upgrade the Gigalixir CLI, run

```
pip3 install -U gigalixir --ignore-installed six
```

31.3 Encryption

All HTTP requests made between your machine and Gigalixir's servers are encrypted.

31.4 Conventions

- No news is good news: If you run a command that produces no output, then the command probably succeeded.
- Exit codes: Commands that succeed will return a 0 exit code, and non-zero otherwise.
- stderr vs stdout: Stderr is used for errors and for log output. Stdout is for the data output of your command.

31.5 Authentication

When you login with your email and password, you receive an API key. This API key is stored in your `~/.netrc` file. Commands generally use your `~/.netrc` file to authenticate with few exceptions.

31.6 Error Reporting

Bugs in the CLI are reported to Gigalixir's error tracking service. Currently, the only way to disable this is by modifying the source code. Pull requests are also accepted!

31.7 Open Source

The Gigalixir CLI is open source and we welcome pull requests. See the gigalixir-cli repository.

CHAPTER 32

How to Set Up Distributed Phoenix Channels

If you have successfully clustered your nodes, then distributed Phoenix channels *just work* out of the box. No need to follow any of the steps described in Running Elixir and Phoenix projects on a cluster of nodes. See more information on how to *cluster your nodes*.

CHAPTER 33

How to Sign Up for an Account

Create an account using the following command. It will prompt you for your email address and password. You will have to confirm your email before continuing. Gigalixir's free tier does not require a credit card, but you will be limited to 1 instance with 0.2GB of memory and 1 postgresql database limited to 10,000 rows.

```
gigalixir signup
```

CHAPTER 34

How to Upgrade an Account

The standard tier offers much more than the free tier, see *Tiers*.

The easiest way to upgrade is through the web interface. Login at https://gigalixir.com/#/signin and click the Upgrade button.

To upgrade with the CLI, first add a payment method

```
gigalixir account:payment_method:set
```

Then upgrade.

```
gigalixir account:upgrade
```

CHAPTER 35

How to Delete an Account

If you just want to make sure you won't be billed anymore, run

```
gigalixir apps
```

And for every app listed, run

```
gigalixir apps:destroy
```

This will make sure you've deleted all domains, databases, etc and you won't be charged in the future.

CHAPTER 36

How to Create an App

To create your app, run the following command. It will also set up a git remote. This must be run from within a git repository folder. An app name will be generated for you, but you can also optionally supply an app name if you wish using `gigalixir create -n $APP_NAME`. There is currently no way to change your app name once it is created. If you like, you can also choose which cloud provider and region using the `--cloud` and `--region` options. We currently support `gcp` in `v2018-us-central1` or `europe-west1` and `aws` in `us-east-1` or `us-west-2`.

```
gigalixir create
```

CHAPTER 37

How to choose a name for your app

Normally, gigalixir generates a unique name for you automatically, but if you want, you can specify your app name. You'll need to *install the CLI* and run something like this

```
gigalixir create -n $APP_NAME
```

That should do it. Once you deploy, you'll be able to access your app from `https://$APP_NAME.gigalixirapp.com`.

CHAPTER 38

How to Delete an App

WARNING!! Deleting an app can not be undone and the name can not be reused.

To delete an app, run

```
gigalixir apps:destroy
```

CHAPTER 39

How to Rename an App

There is no way to rename an app, but you can delete it and then create a new one. Remember to migrate over your configs.

CHAPTER 40

How to Deploy an App

Deploying an app is done using a git push, the same way you would push code to github. For more information about how this works, see *life of a deploy*.

```
git push gigalixir master
```

CHAPTER 41

How to Get Zero-Downtime Deploys

All deploys are automatically zero downtime. No need to do anything. The only exception is if your app serves really long-running requests, like over 30s. During a rolling restart, the old version in terminated about 30 seconds after the new version is healthy. If you need to keep those long-running requests safe, consider *hot upgrades*.

CHAPTER 42

How to Deploy a Branch

To deploy a local branch, `my-branch`, run

```
git push gigalixir my-branch:master
```

CHAPTER 43

How to Set Up a Staging Environment

To set up a separate staging app and production app, you'll need to create another gigalixir app. To do this, first rename your current gigalixir git remote to staging.

```
git remote rename gigalixir staging
```

Then create a new app for production

```
gigalixir create
```

If you like, you can also rename the new app remote.

```
git remote rename gigalixir production
```

From now on, you can run this to push to staging.

```
git push staging master
```

And this to push to production

```
git push production master
```

You'll probably also want to check all your environment variables and make sure they are set probably for production and staging. Also, generally speaking, it's best to use `prod.exs` for both production and staging and let environment variables be the only thing that varies between the two environments. This way staging is as close a simulation of production as possible. If you need to convert any configs into environment variables, use `"${MYVAR}"`.

CHAPTER 44

How to Set Up Continuous Integration (CI/CD)?

Since deploys are just a normal `git push`, Gigalixir should work with any CI/CD tool out there. For Travis CI, put something like this in your `.travis.yml`

```
script:
  - git remote add gigalixir https://$GIGALIXIR_EMAIL:$GIGALIXIR_API_KEY@git.
↪gigalixir.com/$GIGALIXIR_APP_NAME.git
  - mix test && git push -f gigalixir HEAD:refs/heads/master
language: elixir
elixir: 1.5.1
otp_release: 20.0
services:
  - postgresql
before_script:
  - PGPASSWORD=postgres psql -c 'create database gigalixir_getting_started_test;' -U␣
↪postgres
```

Be sure to replace `gigalixir_getting_started_test` with your test database name configured in your `test.exs` file along with your db username and password.

In the Travis CI Settings, add a `GIGALIXIR_EMAIL` environment variable, but be sure to URI encode it e.g. `foo%40gigalixir.com`.

Add a `GIGALIXIR_API_KEY` environment variable which you can find in your `~/.netrc` file e.g. `b9fbde22-fb73-4acb-8f74-f0aa6321ebf7`.

Finally, add a `GIGALIXIR_APP_NAME` environment variable with the name of your app e.g. `real-hasty-fruitbat`

Using GitLab CI or any other CI/CD service should be very similar. For an example GitLab CI yaml file, see this .gitlab-ci.yml file.

Using GitHub Actions is also similar. For example, see https://gist.github.com/jesseshieh/7b231370874445592a40bf5ed6961460

Using CircleCI is also similar. For an example, see this config.yml.

If you want to automatically run migrations on each automatic deploy, you have two options

1. (Recommended) Use a Distillery pre-start boot hook by following https://github.com/bitwalker/distillery/blob/master/docs/guides/running_migrations.md and https://github.com/bitwalker/distillery/blob/master/docs/extensibility/boot_hooks.md
2. Install the gigalixir CLI in your CI environment and run `gigalixir ps:migrate`. For example,

```
# install gigalixir-cli
sudo apt-get install -y python3 python3-pip
pip3 install gigalixir

# deploy
gigalixir login -e "$GIGALIXIR_EMAIL" -p "$GIGALIXIR_PASSWORD" -y
gigalixir git:remote $GIGALIXIR_APP_NAME
git push -f gigalixir HEAD:refs/heads/master
# some code to wait for new release to go live

# set up ssh so we can migrate
mkdir ~/.ssh
printf "Host *\n StrictHostKeyChecking no" > ~/.ssh/config
echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa

# migrate
gigalixir ps:migrate -a $GIGALIXIR_APP_NAME
```

CHAPTER 45

How to Set Up Review Apps (Feature branch apps)

Review Apps let you run a new instance for every branch and tear them down after the branch is deleted. For GitLab CI/CD Review Apps, all you have to do is create a `.gitlab-ci.yml` file that looks something like this one.

Be sure to create CI/CD secrets for `GIGALIXIR_EMAIL`, `GIGALIXIR_PASSWORD`, and `GIGALIXIR_APP_NAME`.

For review apps run on something other than GitLab, the setup should be very similar.

CHAPTER 46

How to Set the Gigalixir Git Remote

If you have a Gigalixir app already created and want to push a git repository to it, set the git remote by running

```
gigalixir git:remote $APP_NAME
```

If you prefer to do it manually, run

```
git remote add gigalixir https://git.gigalixir.com/$APP_NAME.git
```

CHAPTER 47

How to Scale an App

You can scale your app by adding more memory and cpu to each container, also called a replica. You can also scale by adding more replicas. Both are handled by the following command. For more information, see *replica sizing*.

```
gigalixir ps:scale --replicas=2 --size=0.6
```

CHAPTER 48

How to Configure an App

All app configuration is done through environment variables. You can get, set, and delete configs using the following commands. Note that setting configs automatically restarts your app.

```
gigalixir config
gigalixir config:set FOO=bar
gigalixir config:unset FOO
```

CHAPTER 49

How to Copy Configuration Variables

```
gigalixir config:copy -s $SOURCE_APP -d $DESTINATION_APP
```

Note, this will copy all configuration variables from the source to the destination. If there are duplicate keys, the destination config will be overwritten. Variables that only exist on the destination app will not be deleted.

CHAPTER 50

Why was my app scaled down to 0?

On the free tier apps are scaled down to 0 if there have been no deploys for 30 days. We send a warning email after 23 days. To prevent this from happening, make sure you either deploy often or upgrade to the standard tier.

CHAPTER 51

How to Hot Configure an App

This feature is still a work in progress.

CHAPTER 52

How to Hot Upgrade an App

To do a hot upgrade, deploy your app with the extra header shown below. You'll need git v2.9.0 for this to work. For information on how to install the latest version of git on Ubuntu, see this stackoverflow question. For more information about how hot upgrades work, see *Life of a Hot Upgrade*.

```
git -c http.extraheader="GIGALIXIR-HOT: true" push gigalixir master
```

CHAPTER 53

How to Rollback an App

To rollback one release, run the following command.

```
gigalixir releases:rollback
```

To rollback to a specific release, find the `version` by listing all releases. You can see which SHA the release was built on and when it was built. This will also automatically restart your app with the new release.

```
gigalixir releases
```

You should see something like this

```
[
  {
    "created_at": "2017-04-12T17:43:28.000+00:00",
    "version": "5",
    "sha": "77f6c2952129ffecccc4e56ae6b27bba1e65a1e3",
    "summary": "Set `DATABASE_URL` config var."
  },
  ...
]
```

Then specify the version when rolling back.

```
gigalixir releases:rollback --version=5
```

The release list is immutable so when you rollback, we create a new release on top of the old releases, but the new release refers to the old slug.

CHAPTER 54

How to Set Up a Custom Domain

After your first deploy, you can see your app by visiting https://\protect\T1\textdollarAPP_NAME.gigalixirapp.com/, but if you want, you can point your own domain such as www.example.com to your app. To do this, run the following command and follow the instructions.

```
gigalixir domains:add www.example.com
```

If you have version 0.27.0 or later of the CLI, you'll be given instructions on what to do next. If not, run `gigalixir domains` and use the `cname` value to point your domain at.

This will do a few things. It registers your fully qualified domain name in the load balancer so that it knows to direct traffic to your containers. It also sets up SSL/TLS encryption for you. For more information on how SSL/TLS works, see *How SSL/TLS Works*.

If your DNS provider does not allow CNAME, which is common for naked/root domains, and you are using the gcp v2018-us-central1 region, the default, you can also use an A record. Use the IP address 35.226.132.161. For gcp europe-west1, use 130.211.67.69. For AWS, unfortunately, you have to use a CNAME so the only option is to change DNS providers. While we have no plans to change these ip addresses, we highly recommend you use CNAMEs if at all possible.

Note that if you want both the naked/root domain and a subdomain such as www, be sure to run *gigalixir domains:add* for each one.

If you need a wildcard domain, feel free to *contact us* and we can help you get set up.

Note that with Phoenix, you may need to change your `check_origin` setting in order for websockets to pass the origin check. See https://hexdocs.pm/phoenix/Phoenix.Endpoint.html#module-runtime-configuration

CHAPTER 55

How to Set Up SSL/TLS

SSL/TLS certificates are set up for you automatically assuming your custom domain is set up properly. Note that your application will continue to be served on http as well as https. If you want to force your users to use https by redirecting any http requests, specify that in your *config/prod.exs*:

```
config :my_app, MyAppWeb.Endpoint,
   force_ssl: [rewrite_on: [:x_forwarded_proto]]
```

This configures your app to check the x-forwarded-proto header set by Gigalixir, and redirect to https, if appropriate.

For more information on how this works internally, see *How SSL/TLS Works*.

CHAPTER 56

How to Tail Logs

You can tail logs in real-time aggregated across all containers using the following command.

```
gigalixir logs
```

CHAPTER 57

How to Forward Logs Externally

If you want to forward your logs to another service such as Timber or PaperTrail, you'll need to set up a log drain. We support HTTPS and syslog drains. To create a log drain, run

```
gigalixir drains:add $URL
# e.g. gigalixir drains:add https://user:$TIMBER_API_KEY@logs.timber.io/sources/
↪$TIMBER_SOURCE_ID/frames
# e.g. gigalixir drains:add syslog+tls://logs123.papertrailapp.com:12345
```

To show all your drains, run

```
gigalixir drains
```

To delete a drain, run

```
gigalixir drains:remove $DRAIN_ID
```

CHAPTER 58

Managing SSH Keys

In order to SSH, run remote observer, remote console, etc, you need to set up your SSH keys. It could take up to a minute for the SSH keys to update in your containers.

```
gigalixir account:ssh_keys:add "$(cat ~/.ssh/id_rsa.pub)"
```

If you don't have an `id_rsa.pub` file, follow this guide to create one.

To view your SSH keys

```
gigalixir account:ssh_keys
```

To delete an SSH key, find the key's id and then run delete the key by id.

```
gigalixir account:ssh_keys:remove $ID
```

CHAPTER 59

How to SSH into a Production Container

If your app is running, but not behaving, SSH'ing in might give you some insight into what is happening. A major caveat, though, is that the app has to be running. If it isn't running, then it isn't passing health checks, and we'll keep restarting the entire container. You won't be able to SSH into a container that is restarting non-stop. If your app isn't running, try taking a look at *Troubleshooting*.

To SSH into a running production container, first, add your public SSH keys to your account. For more information on managing SSH keys, see *Managing SSH Keys*.

```
gigalixir account:ssh_keys:add "$(cat ~/.ssh/id_rsa.pub)"
```

Then use the following command to SSH into a live production container. If you are running multiple containers, this will put you in a random container. We do not yet support specifying which container you want to SSH to. In order for this work, you must add your public SSH keys to your account.

```
gigalixir ps:ssh
```

CHAPTER 60

How to specify SSH key or other SSH options

The `-o` option lets you pass in arbitrary options to `ssh`. Something like this will let you specify which SSH key to use.

```
gigalixir ps:ssh -o "-i ~/.ssh/id_rsa"
```

CHAPTER 61

How to List Apps

To see what apps you own and information about them, run the following command. This will only show you your desired app configuration. To see the actual status of your app, see *How to Check App Status*.

```
gigalixir apps
```

CHAPTER 62

How to List Releases

Each time you deploy or rollback a new release is generated. To see all your previous releases, run

```
gigalixir releases
```

CHAPTER 63

How to Change or Reset Your Password

With the web interface, visit https://gigalixir.com/#/signin-help

With the CLI, run

```
gigalixir account:password:change
```

If you forgot your password, send a reset token to your email address by running the following command and following the instructions in the email.

```
gigalixir account:password:reset
```

CHAPTER 64

How to Change My Email Address

Contact us and we'll help you out.

CHAPTER 65

How to Resend the Confirmation Email

With the web interface, visit https://gigalixir.com/#/signin-help

With the CLI, run

```
gigalixir account:confirmation:resend
```

CHAPTER 66

How to Change Your Credit Card

To change your credit card, run

```
gigalixir account:payment_method:set
```

CHAPTER 67

How to Delete your Account

There is currently no way to completely delete an account. We are working on implementing this feature. You can delete apps though. See *How to Delete an App*.

CHAPTER 68

How to Restart an App

```
gigalixir ps:restart
```

For hot upgrades, See *How to Hot Upgrade an App*. We are working on adding custom health checks.

Restarts should be zero-downtime. See *How to Set Up Zero-Downtime Deploys*.

CHAPTER 69

How to Set Up Zero-Downtime Deploys

Normally, there is nothing you need to do to have zero-downtime deploys. The only caveat is that health checks are currently done by checking if tcp port 4000 is listening. If your app opens the port before it is ready, then it may start receiving traffic before it is ready to serve it. In most cases, with Phoenix, this isn't a problem.

One downside of zero-downtime deploys is that they make deploys slower. What happens during a deploy is

1. Spawn a new instance
2. Wait for health check on the new instance to pass
3. Start sending traffic to the new instance
4. Stop sending traffic to the old instance
5. Wait 30 seconds for old instance is finish processing requests
6. Terminate the old instance
7. Repeat for every instance

Although you should see your new code running within a few seconds, the entire process takes over 30 seconds per instance so if you have a lot of instances running, this could take a long time.

Heroku opts for faster deploys and restarts instead of zero-downtime deploys.

CHAPTER 70

How to Run Jobs

There are many ways to run one-off jobs and tasks. You can run them in the container your app is running or you can spin up a new container that runs the command and then destroys itself.

To run a command in your app container, run

```
gigalixir ps:run $COMMAND
# if you're using distillery, you'll probably want $COMMAND to be something like␣
↪:bash:`bin/app eval 'IO.inspect Node.self'`
# if you're using mix, you'll probably want $COMMAND to be something like :bash:`mix␣
↪ecto.migrate`
```

To run a command in a separate container, run

```
gigalixir run $COMMAND
# if you're using distillery, you'll probably want $COMMAND to be something like␣
↪:bash:`bin/app eval 'IO.inspect Node.self'`
# if you're using mix, you'll probably want $COMMAND to be something like :bash:`mix␣
↪ecto.migrate`
```

The task is not run on the same node that your app is running in. Jobs are killed after 5 minutes.

If you're using the distillery, note that because we start a separate container to run the job, if you need any applications started such as your `Repo`, use `Application.ensure_all_started/2`. Also, be sure to stop all applications when done, otherwise your job will never complete and just hang until it times out.

Distillery commands currently do not support passing arguments into the job.

We prepend `Elixir.` to your module name to let the BEAM virtual machine know that you want to run an Elixir module rather than an Erlang module. The BEAM doesn't know the difference between Elixir code and Erlang code once it is compiled down, but compiled Elixir code is namespaced under the Elixir module.

The size of the container that runs your job will be the same size as the app containers and billed the same way, based on replica-size-seconds. See, *Pricing Details*.

CHAPTER 71

How to Reset your API Key

If you lost your API key or it has been stolen, you can reset it by running

```
gigalixir account:api_key:reset
```

Your old API key will no longer work and you may have to login again.

CHAPTER 72

How to Log Out

```
gigalixir logout
```

CHAPTER 73

How to Log In

```
gigalixir login
```

This modifies your ~/.netrc file so that future API requests will be authenticated. API keys never expire, but can be revoked.

CHAPTER 74

How to provision a Free PostgreSQL database

IMPORTANT: Make sure you set your `pool_size` in `prod.exs` to 2 beforehand. The free tier database only allows limited connections.

The following command will provision a free database for you and set your `DATABASE_URL` environment variable appropriately.

```
gigalixir pg:create --free
```

List databases by running

```
gigalixir pg
```

Delete by running

```
gigalixir pg:destroy -d $DATABASE_ID
```

You can only have one database per app because otherwise managing your `DATABASE_URL` variable would become trickier.

In the free tier, the database is free, but it is really not suitable for production use. It is a multi-tenant postgres database cluster with shared CPU, memory, and disk. You are limited to 2 connections, 10,000 rows, and no backups. Idle connections are terminated after 5 minutes. If you want to upgrade your database, you'll have to migrate the data yourself. For a complete feature comparison see *Tiers*.

For information on upgrading your account, see *How to Upgrade an Account*.

CHAPTER 75

How to provision a Standard PostgreSQL database

The following command will provision a database for you and set your `DATABASE_URL` environment variable appropriately.

```
gigalixir pg:create --size=0.6
```

It takes a few minutes to provision. You can check the status by running

```
gigalixir pg
```

You may also want to adjust your pool_size. We recommend setting the pool size to (M-6)/(n+1) where M is the max connections and n is the num app replicas. We subtract 6 because cloud sql will sometimes, but rarely, use 6 for maintenance purposes. We use n+1 because rolling deploys will temporarily have an extra replica during the transition. For example, if you are running a size 0.6 database with 1 app replica, the pool size should be (25-6)/(1+1)=9.

You can only have one database per app because otherwise managing your `DATABASE_URL` variable would become trickier.

Under the hood, we use Google's Cloud SQL which provides reliability, security, and automatic backups. For more information, see Google Cloud SQL for PostgreSQL Documentation.

CHAPTER 76

How to upgrade a Free DB to a Standard DB

If you started out with a free tier database and then upgraded to the standard tier, we highly recommend you migrate to a standard tier database. The standard tier databases support encryption, backups, extensions, and dedicated cpu, memory, & disk. There are no row limits, connection limits*, and they are automatically scalable.

Unfortunately, we can't automatically migrate your free tier db to a standard tier db. You'll have to

1. `pg_dump` the free database
2. Delete the free database with `gigalixir pg:destroy --help`. Note postgres may make you scale down to 0 app replicas to do this so you'll have some downtime.
3. Create the standard tier database with `gigalixir pg:create`.
4. Restore the data with `psql` or `pg_restore`. You can find the url to use with `gigalixir pg` once the standard tier database is created.

Please don't hesitate to *contact us* if you need help.

- Databases still have connection limits based on Google Cloud SQL limits. See https://cloud.google.com/sql/docs/postgres/quotas#fixed-limits

CHAPTER 77

How to scale a database

To change the size of your database, run

```
gigalixir pg:scale -d $DATABASE_ID --size=1.7
```

You can find your database id by running

```
gigalixir pg
```

Supported sizes include 0.6, 1.7, 4, 8, 16, 32, 64, and 128. For more information about databases sizes, see *Database Sizes & Pricing*.

CHAPTER 78

How to restore a database backup

We use Cloud SQL under the hood which takes automatic backups every day and keeps 7 backups available. For more, see https://cloud.google.com/sql/docs/postgres/backup-recovery/backups

First, get your database id by running

```
gigalixir pg
```

View what backups you have available by running

```
gigalixir pg:backups -d $DATABASE_ID
```

Note: we required the database_id even though we could probably detect it automatically because these are sensitive operations and we prefer to be explicit.

Find the backup id you want and run

```
gigalixir pg:backups:restore -d $DATABASE_ID -b $BACKUP_ID
```

This can take a while. Sometimes over ten minutes. To check the status, run

```
gigalixir pg
```

CHAPTER 79

How to restart a database

Contact us and we'll help you out. Only standard tier databases can be restarted.

CHAPTER 80

How to delete a database

WARNING!! Deleting a database also deletes all of its backups. Please make sure you backup your data first.

To delete a database, run

```
gigalixir pg:destroy -d $DATABASE_ID
```

CHAPTER 81

How to install a Postgres Extension

Note: Free Databases do not support extensions. See *Tiers*.

First, make sure Google Cloud SQL supports your extension by checking their list of extensions. If it is supported, find your database url by running

```
gigalixir pg
```

Then, get a psql console into your database

```
psql $DATABASE_URL
```

Then, install your extension

```
CREATE EXTENSION foo;
```

CHAPTER 82

How to Connect a Database

If you followed the *Getting Started Guide*, then your database should already be connected. If not, connecting to a database is done no differently from apps running outside Gigalixir. We recommend you set a DATABASE_URL config and configure your database adapter accordingly to read from that variable. In short, you'll want to add something like this to your `prod.exs` file.

```
config :gigalixir_getting_started, GigalixirGettingStarted.Repo,
  adapter: Ecto.Adapters.Postgres,
  url: {:system, "DATABASE_URL"},
  database: "",
  ssl: true,
  pool_size: 2
```

Replace `:gigalixir_getting_started` and `GigalixirGettingStarted` with your app name. Then, be sure to set your `DATABASE_URL` config with something like this. For more information on setting configs, see *How to Configure an App*. If you provisioned your database using, *How to provision a Standard PostgreSQL database*, then `DATABASE_URL` should be set for you automatically once the database is provisioned. Otherwise,

```
gigalixir config:set DATABASE_URL="ecto://user:pass@host:port/db"
```

If you need to provision a database, Gigalixir provides Databases-as-a-Service. See *How to provision a Standard PostgreSQL database*. If you prefer to provision your database manually, follow *How to set up a Google Cloud SQL PostgreSQL database*.

82.1 How to manually set up a Google Cloud SQL PostgreSQL database

Note: You can also use Amazon RDS, but we do not have instructions provided yet.

1. Navigate to https://console.cloud.google.com/sql/instances and click "Create Instance".
2. Select PostgreSQL and click "Next".
3. Configure your database.

   a. Choose any instance id you like.
   b. Choose us-central1 as the Region.
   c. Choose how many cores, memory, and disk.
   d. In "Default user password", click "Generate" and save it somewhere secure.
   e. In "Authorized networks", click "Add network" and enter "0.0.0.0/0" in the "Network" field. It will be encrypted with TLS and authenticated with a password so it should be okay to make the instance publicly accessible. Click "Done".
4. Click "Create".
5. Wait for the database to create.
6. Make note of the database's external ip. You'll need it later.
7. Click on the new database to see instance details.
8. Click on the "Databases" tab.
9. Click "Create database".
10. Choose any name you like, remember it, and click "Create".
11. Run

```
gigalixir config:set DATABASE_URL="ecto://postgres:$PASSWORD@$EXTERNAL_IP:5432/
↪$DB_NAME"
```

    with $PASSWORD, $EXTERNAL_IP, and $DB_NAME replaced with values from the previous steps.
12. Make sure you have `ssl:true` in your `prod.exs` database configuration. Cloud SQL supports TLS out of the boxso your database traffic should be encrypted.

We hope to provide a database-as-a-service soon and automate the process you just went through. Stay tuned.

CHAPTER 83

How to Run Migrations

If you deployed your app without distillery or Elixir releases (mix mode), you can run migrations as a job in a new container with

```
gigalixir run mix ecto.migrate
```

If you deployed your app as a distillery release or elixir release, `mix` isn't available. We try to make it easy by providing a special command, but the command runs on your existing app container so you'll need to make sure your app is running first and set up SSH keys.

```
gigalixir account:ssh_keys:add "$(cat ~/.ssh/id_rsa.pub)"
```

Then run

```
gigalixir ps:migrate
```

This command runs your migrations in a remote console directly on your production node. It makes some assumptions about your project so if it does not work, please *contact us for help*.

If you are running an umbrella app, you will probably need to specify which "inner app" within your umbrella to migrate. Do this by passing the `--migration_app_name` flag like so

```
gigalixir ps:migrate --migration_app_name=$MIGRATION_APP_NAME
```

If you want to run migrations automatically before each deploy, we suggest using a distillery pre-start boot hook by following https://github.com/bitwalker/distillery/blob/master/docs/guides/running_migrations.md and https://github.com/bitwalker/distillery/blob/master/docs/extensibility/boot_hooks.md

If you aren't running distillery, you can try modifying your `Procfile` to something like this

```
web: mix ecto.migrate && elixir --name $MY_NODE_NAME --cookie $MY_COOKIE -S mix␣
↪phoenix.server
```

For more details, see *Can I use a custom Procfile?*.

When running `gigalixir ps:migrate`, sometimes the migration doesn't do exactly what you want. If you need to tweak the migration command to fit your situation, all `gigalixir ps:migrate` is doing is dropping into a remote_console and running the following. For information on how to open a remote console, see *How to Drop into a Remote Console*.

```
repo = List.first(Application.get_env(:gigalixir_getting_started, :ecto_repos))
app_dir = Application.app_dir(:gigalixir_getting_started, "priv/repo/migrations")
Ecto.Migrator.run(repo, app_dir, :up, all: true)
```

So for example, a tweak you might make is, if you have more than one app, you may not want to use `List.first` to find the app that contains the migrations.

If you have a chicken-and-egg problem where your app will not start without migrations run, and migrations won't run without an app running, you can try the following workaround on your local development machine. This will run migrations on your production database from your local machine using your local code.

```
MIX_ENV=prod DATABASE_URL="$YOUR_PRODUCTION_DATABASE_URL" mix ecto.migrate
```

CHAPTER 84

How to reset the database?

First, *drop into a remote console* and run this to "down" migrate. You may have to tweak the command depending on what your app is named and if you're running an umbrella app.

```
Ecto.Migrator.run(MyApp.Repo, Application.app_dir(:my_app, "priv/repo/migrations"),␣
↪:down, [all: true])
```

Then run this to "up" migrate.

```
Ecto.Migrator.run(MyApp.Repo, Application.app_dir(:my_app, "priv/repo/migrations"),␣
↪:up, [all: true])
```

CHAPTER 85

How to run seeds?

Running seeds in production is usually a one-time job, so our recommendation is to *drop into a remote console* and run commands manually. If you have a `seeds.exs` file, you can follow the Distillery migration guide and run something like this in your remote console.

```
seed_script = Path.join(["#{:code.priv_dir(:myapp)}", "repo", "seeds.exs"])
Code.eval_file(seed_script)
```

CHAPTER 86

How to Drop into a Remote Console

To get a console on a running production container, first, add your public SSH keys to your account. For more information on managing SSH keys, see *Managing SSH Keys*.

```
gigalixir account:ssh_keys:add "$(cat ~/.ssh/id_rsa.pub)"
```

Then run this command to drop into a remote console.

```
gigalixir ps:remote_console
```

CHAPTER 87

How to Run Distillery Commands

Since we use Distillery to build releases, we also get all the commands Distillery provides such as ping, rpc, command, and eval. *Launching a remote console* is just a special case of this. To run a Distillery command, run the command below. For a complete list of commands, see Distillery's boot.eex.

```
gigalixir ps:distillery $COMMAND
```

CHAPTER 88

How to Check App Status

To see how many replicas are actually running in production compared to how many are desired, run

```
gigalixir ps
```

CHAPTER 89

How to Check Account Status

To see things like which account you are logged in as, what tier you are on, and how many credits you have available, run

```
gigalixir account
```

CHAPTER 90

How to Launch a Remote Observer

To connect a remote observer, you need to be using Distillery. See *Mix vs Distillery vs Elixir Releases*.

In order to run a remote observer, you need to set up your SSH keys. It could take up to a minute for the SSH keys to update in your containers.

```
gigalixir account:ssh_keys:add "$(cat ~/.ssh/id_rsa.pub)"
```

Because Observer runs on your local machine and connects to a production node by joining the production cluster, you first have to have clustering set up. You don't have to have multiple nodes, but you need to follow the instructions in *Clustering Nodes*.

You also need to have `runtime_tools` in your application list in your `mix.exs` file. Phoenix 1.3 and later adds it by default, but you have to add it yourself in Phoenix 1.2.

Your local machine also needs to have `lsof`.

You should also make sure your app has enough memory. Even though observer itself is running on your local machine, the remote machine still needs quite a bit of memory. For a basic app, make sure you have at least 500mb memory (size 0.5).

Then, to launch observer and connect it to a production node, run

```
gigalixir ps:observer
```

and follow the instructions. It will prompt you for your local sudo password so it can modify iptables rules. This connects to a random container using consistent hashing. We don't currently allow you to specify which container you want to connect to, but it will connect to the same container each time based on a hash of your ip address.

CHAPTER 91

How to see the current period's usage

To see how many replica-size-seconds you've used so far this month, run

```
gigalixir account:usage
```

The amount you see here has probably not been charged yet since we do that at the end of the month.

CHAPTER 92

How to see previous invoices

To see all your previous period's invoices, run

```
gigalixir account:invoices
```

CHAPTER 93

Teams & Organizations

If you work in a team, you'll probably want to collaborate with other users. With gigalixir's access permissions, you can grant access using the commands below. They'll be able to deploy & rollback, manage configs, ssh, remote_console, observer, hot upgrade, and scale.

First, they need to sign up for their own gigalixir account. Then run the command below to give them access.

```
gigalixir access:add $USER_EMAIL
```

To see, who has access, run

```
gigalixir access
```

To deny access to a user, run

```
gigalixir access:remove $USER_EMAIL
```

If you don't have access to the CLI and want to modify access, *contact us* and we'll help you out.

The "owner", the user who created the app, is responsible for the bill each month.

For organizations, we recommend creating an "organization account" that is upgraded to the standard tier and has the billing information on file. Then create individual accounts for all developers and grant access to all contributors.

CHAPTER 94

How do I change the owner of my app?

No problem. *Contact us* and we'll help you out.

CHAPTER 95

How to deploy a Ruby app

```
gigalixir login
git clone https://github.com/heroku/ruby-getting-started.git
cd ruby-getting-started
APP=$(gigalixir create)
git push gigalixir master
curl https://$APP.gigalixirapp.com/
```

CHAPTER 96

How do I use webpack, yarn, bower, gulp, etc instead of brunch?

You can use a custom compile script. For more details, see https://github.com/gjaldon/heroku-buildpack-phoenix-static#compile Here is an example script that we've used for webpack.

```
cd $assets_dir
node_modules/.bin/webpack -p

cd $phoenix_dir
mix "${phoenix_ex}.digest"
```

CHAPTER 97

How to specify which release, environment, or profile to build

97.1 Distillery

If you have multiple releases defined in `rel/config.exs`, which is common for umbrella apps, you can specify which release to build by setting a config variable on your app that controls the options passed to *mix distillery.release*. For example, you can pass the *–profile* option using the command below.

```
gigalixir config:set GIGALIXIR_RELEASE_OPTIONS="--profile=$RELEASE_NAME:$RELEASE_
↪ENVIRONMENT"
```

With this config variable set on each of your gigalixir apps, when you deploy the same repo to each app, you'll get a different release.

If you have multiple Phoenix apps in the umbrella, instead of deploying each as a separate distillery release, you could also consider something like this master_proxy to proxy requests to the two apps.

97.2 Elixir Releases

If you want to pass options to `mix release` such as the release name, you can specify options with the `GIGALIXIR_RELEASE_OPTIONS` env var.

For example, to build a different release other than the default, run

```
gigalixir config:set GIGALIXIR_RELEASE_OPTIONS="my-release"
```

CHAPTER 98

How do I use a private hex dependency?

First, take a look at the following page and generate an auth key for your org https://hex.pm/docs/private#authenticating-on-ci-and-build-servers

Add something like this to your `elixir_buildpack.config` file

```
hook_pre_fetch_dependencies="mix hex.organization auth myorg --key ${HEX_ORG_AUTH}"
```

Then run

```
gigalixir config:set HEX_ORG_AUTH="authkeyhere"
```

CHAPTER 99

How do I use a private git dependency?

If you want to use a private git repository as a dependency in `mix.exs`, our recommended approach is to use the netrc buildpack found at https://github.com/timshadel/heroku-buildpack-github-netrc

To use the buildpack, insert it in your `.buildpacks` file above the Elixir and Phoenix buildpacks. For example, if you are using distillery, your `.buildpacks` file will look like this

```
https://github.com/timshadel/heroku-buildpack-github-netrc.git
https://github.com/HashNuke/heroku-buildpack-elixir
https://github.com/gjaldon/heroku-buildpack-phoenix-static
https://github.com/gigalixir/gigalixir-buildpack-distillery.git
```

Next, create a personal access token by following https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/

Just make sure you give the token "repo" access so that it can access your private repository.

Add your personal access token as a config var by running

```
gigalixir config:set -a $APP_NAME GITHUB_AUTH_TOKEN="$GITHUB_TOKEN"
```

The last step is to add the dependency to your `mix.exs` file. Add it as you would any other git dependency, but be sure you use the https url and not the ssh url. For example,

```
{:foo, git: "https://github.com/jesseshieh/foo.git", override: true}
```

That should be it.

Alternatively, you could also put your github username and personal access token directly into the git url, but it's generally not a good idea to check in secrets to source control. You could use `System.get_env` interpolated inside the git url, but then you run the risk of the secrets getting saved to `mix.lock`.

CHAPTER 100

How can I get the current SHA my app is running?

There are a number of environment variables set in your app container. `SOURCE_VERSION` contains your current SHA.

CHAPTER 101

What environment variables are available to my app?

SOURCE_VERSION contains the current SHA

HOST_INDEX contains the index of the replica. The hostname for each replica is randomly generated which can be a problem for services like DataDog and NewRelic who charge by the host. We also keep a sort of ordered list of your replicas that you can use to report hostnames to keep your number of hosts low. Each replica currently running will have a different HOST_INDEX, but once a replica is terminated, its HOST_INDEX can be re-used in another replica.

APP_NAME contains your gigalixir app name.

APP_KEY contains the app specific key you need to fetch information about your app from inside the replica. You probably don't need to use this unless you're doing something really low level, but it's there if you need it.

ERLANG_COOKIE contains a randomly generated UUID that we use as your erlang distribution cookie. We set it for you automatically and it's used in your default vm.args file so you don't need to mess with anything, but it's here if you should want to use it.

LOGPLEX_TOKEN contains the app specific token we use to send your app logs to logplex. Logplex is our central log router which handles aggregating, draining, and tailing your logs. You can use this if you want to do something custom with logs that can't be done by printing to stdout from your app.

MY_POD_IP contains your replica/container/pod's ip address.

PORT contains the port your app needs to listen on to pass health checks and receive traffic. It is almost always 4000, but we reserve the right to change or randomize it.

SECRET_KEY_BASE contains a randomly generated string that we use as your Elixir app's secret key base.

HOME contains the location of your app's home directly. It is almost always /app, but we reserve the right to change it.

CHAPTER 102

Does Gigalixir have any web hooks?

We haven't built-in any web hooks, but most of what you need can be accomplished with buildpacks at build time and distillery hooks or modifying your Procfile.

To hit a web hook when building your app, you can use https://github.com/jesseshieh/buildpack-webhook

For runtime prestart hooks, see https://hexdocs.pm/distillery/extensibility/boot_hooks.html

Or if you aren't using distillery, see *Can I use a custom Procfile?*. You can add any command you like.

CHAPTER 103

Can I choose my operating system, stack, or image?

We have 3 stacks you can choose from: gigalixir-14, gigalixir-16, and gigalixir-18. These stacks are based on Heroku's cedar-14, heroku-16, and heroku-18, respectively which are based on Ubuntu 14, 16, and 18 respectively. gigalixir-18 is the default.

You can choose your stack when you create your app with

```
gigalixir create --stack gigalixir-16
```

or you can change it later on with

```
gigalixir stack:set --stack gigalixir-18
```

You can see what stack you are running with `gigalixir apps:info` or `gigalixir ps`.

For information about what packages are available in each stack, see https://devcenter.heroku.com/articles/stack-packages as well as the Dockerfiles at https://github.com/gigalixir/gigalixir-run

CHAPTER 104

How do I enable bash auto-completion?

Add the following to your `.bashrc` file and restart your shell.

```
eval "$(_GIGALIXIR_COMPLETE=source gigalixir)"
```

CHAPTER 105

How secure is Gigalixir?

Gigalixir takes security very, very seriously.

1. Every app exists in its own Kubernetes namespaces and we use Kubernetes role-based access controls to ensure no other apps have access to your app or its metadata.
2. Your build environment is fully isolated using Docker containers.
3. Your slugs are authenticated using Signed URLs.
4. All API endpoints are authenticated using API keys instead of your password. API keys can be invalidated at any time by regenerating a new one.
5. Remote console and remote observer use SSH tunnels to secure traffic.
6. Erlang does not encrypt distribution traffic between your nodes by default, but you can set it up to use SSL. For an extra layer of security, we route distribution traffic directly to each node so no other apps can sniff the traffic.
7. We use Stripe to manage payment methods so Gigalixir never knows your credit card number.
8. Passwords and app configs are encrypted at rest using Cloak.
9. Traffic between Gigalixir services and components is TLS encrypted.

CHAPTER 106

Are there any benchmarks?

Take a look at our benchmark data.

CHAPTER 107

Money-back Guarantee

If you are unhappy for any reason within the first 31 days, contact us to get a refund up to $75. Enough to run a 3 node cluster for 31 days.

CHAPTER 108

Indices and Tables

- genindex
- modindex
- search