
GIGALIXIR Documentation

Release 0.15.0

Invisible Software, Inc.

Sep 14, 2017

Contents:

1	What is Gigalixir?	3
2	Getting Started Guide	5
2.1	Prerequisites	5
2.2	Install the Command-Line Interface	5
2.3	Create an Account	6
2.4	Log In	6
2.5	Prepare Your App	6
2.6	Set Up App for Deploys	6
2.7	Deploy!	7
2.8	Provision a Database	7
2.9	What's Next?	7
3	Modifying an Existing App to Run on Gigalixir	9
3.1	Required Modifications	9
3.1.1	Install Distillery to Build Releases	9
3.1.2	Specify Buildpacks to Compile and Build Releases	9
3.1.3	Configuration and Secrets	10
3.2	Optional Modifications	10
3.2.1	Set up Node Clustering with Libcluster	11
3.2.2	Set Up Migrations	11
3.2.3	Set Up Hot Upgrades with Git v2.9.0	11
4	Use Cases	13
4.1	I'm learning Elixir and need access to all Elixir's features	13
4.2	I'm using Heroku, but I've run into limitations	13
4.3	I'm using AWS, but I'm spending too much time on infrastructure	13
5	Known Issues	15
6	How Does Gigalixir Work?	17
6.1	Components	18
6.2	Concepts	19
6.3	Life of a Deploy	19
6.4	How SSL/TLS Works	20
6.5	Cleaning Your Cache	20
6.6	Life of a Hot Upgrade	20

7	How do I specify my Elixir, Erlang, Node, NPM, etc versions?	21
8	Frequently Asked Questions	23
8.1	<i>Do you support umbrella apps?</i>	23
8.2	<i>Do you support Phoenix 1.3?</i>	23
8.3	<i>What is Elixir? What is Phoenix?</i>	23
8.4	<i>How is Gigalixir different from Heroku and Deis Workflow?</i>	24
8.5	<i>I thought you weren't supposed to SSH into docker containers!?</i>	25
8.6	<i>Why do you download the slug on startup instead of including the slug in the Docker image?</i>	25
8.7	<i>How do I add worker processes?</i>	25
8.8	<i>What if Gigalixir shuts down?</i>	25
8.9	<i>My git push was rejected</i>	26
9	Clustering Nodes	27
10	Tiers	29
11	Gigalixir vs Heroku Feature Comparison	31
12	Pricing Details	33
13	Replica Sizing	35
14	Releases	37
15	Limits	39
16	Monitoring	41
17	Using Environment Variables in your App	43
18	Troubleshooting	45
19	Support/Help	47
20	The Gigalixir Command-Line Interface	49
20.1	Installation	49
20.2	Upgrade	49
20.3	Encryption	49
20.4	Conventions	49
20.5	Authentication	50
20.6	Error Reporting	50
20.7	Open Source	50
21	How to Set Up Distributed Phoenix Channels	51
22	How to Sign Up for an Account	53
23	How to Upgrade an Account	55
24	How to Create an App	57
25	How to Deploy an App	59
26	How to Deploy a Branch	61
27	How to Set Up a Staging Environment	63

28	How to Set Up Continuous Integration (CI/CD)?	65
29	How to Set the Gigalixir Git Remote	67
30	How to Scale an App	69
31	How to Configure an App	71
32	How to Hot Configure an App	73
33	How to Hot Upgrade an App	75
34	How to Rollback an App	77
35	How to Set Up a Custom Domain	79
36	How to Set Up SSL/TLS	81
37	How to Tail Logs	83
38	How to Forward Logs Externally	85
39	Managing SSH Keys	87
40	How to SSH into a Production Container	89
41	How to List Apps	91
42	How to List Releases	93
43	How to Change or Reset Your Password	95
44	How to Change Your Credit Card	97
45	How to Delete an App	99
46	How to Delete your Account	101
47	How to Restart an App	103
48	How to Run Jobs	105
49	How to Reset your API Key	107
50	How to Log Out	109
51	How to Log In	111
52	How to provision a PostgreSQL database	113
53	How to scale a database	115
54	How to delete a database	117
55	How to install a Postgres Extension	119
56	Database Sizes & Pricing	121

57	How to Connect a Database	123
57.1	How to manually set up a Google Cloud SQL PostgreSQL database	123
58	How to Run Migrations	125
59	How to run seeds?	127
60	How to Drop into a Remote Console	129
61	How to Run Distillery Commands	131
62	How to Check App Status	133
63	How to Check Account Status	135
64	How to Launch a Remote Observer	137
65	How to see the current period's usage	139
66	How to see previous invoices	141
67	How secure is Gigalixir?	143
68	Money-back Guarantee	145
69	Indices and Tables	147

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*.

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 your 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.

Prerequisites

1. `python2.7`, not `python3`. Most OSes already have this installed.
2. `pip`. For help, take a look at the [pip documentation](#).
3. `git`. For help, take a look at the [git documentation](#).
4. Linux or OS X.

For example, on Ubuntu, run

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

Install the Command-Line Interface

Next install, the command-line interface. Gigalixir currently does not have a web interface. We want the command-line to be a first-class citizen so that you can build scripts and tools easily.

```
pip install gigalixir
```

Verify by running

```
gigalixir --help
```

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
```

Log In

Next, log in. This will grant you an api key which expires in 365 days. It will also optionally modify your ~/.netrc file so that all future commands are authenticated.

```
gigalixir login
```

Verify by running

```
gigalixir account
```

Prepare Your App

If you have an existing app or want to use `mix phoenix.new`, follow the steps in *Modifying an Existing App to Run on Gigalixir*. If you are starting a project from scratch, the easiest way to get started is to clone the `gigalixir-getting-started` repo.

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

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. There is currently no way to change your app name.

```
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
```

Deploy!

Finally, build and deploy.

```
git push gigalixir master
```

Wait a minute or two since this is the first deploy, then verify by running

```
curl https://$APP_NAME.gigalixirapp.com/
```

Provision a Database

Your app does not have a database yet, let's create one.

```
gigalixir create_database $APP_NAME
```

This may take a few minutes to become AVAILABLE. Run this to check the status.

```
gigalixir databases $APP_NAME
```

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

```
gigalixir configs $APP_NAME
```

What's Next?

- [How to Tail Logs](#)
- [How to Configure an App](#)
- [How to Scale an App](#)
- [How to Restart an App](#)
- [How to Rollback an App](#)
- [How to Run Migrations](#)
- [How to Drop into a Remote Console](#)
- [How to Launch a Remote Observer](#)
- [How to Hot Upgrade an App](#)

Modifying an Existing App to Run on Gigalixir

Whether you have an existing app or you just ran `mix phoenix.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 4000, you should be good.

Required Modifications

These modifications are required to run on Gigalixir, but features such as node clustering probably won't work unless you make some optional modifications described in the next section.

Install Distillery to Build Releases

Distillery is currently the only supported release tool. We assume you have followed the [Distillery installation instructions](#). We use Distillery instead of bundling up your source code is to support hot upgrades.

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

```
{:distillery, "~> 1.0.0"}
```

Then, run

```
mix deps.get
mix release.init
```

Note, there is a [known issue](#) right now with Distillery 1.3.5 with Elixir 1.3.1, but a fix should be released soon. In the meantime, if you are on Elixir 1.3.1, try using Distillery 1.0.0.

Specify Buildpacks to Compile and Build Releases

We rely on buildpacks to compile and build your release. Create a `.buildpacks` file with the following contents.

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

If you *really* want, the `gigalixir-buildpack-clean-cache` is optional if you know you will never want to clean your Gigalixir build cache. Also, `heroku-buildpack-phoenix-static` is optional if you do not have phoenix static assets. For more information about buildpacks, see *Life of a Deploy*.

Configuration and Secrets

By default, Phoenix creates a `prod.secret.exs` file to store secrets. If you want to continue using `prod.secret.exs` you'll have to commit it to version control so we can bundle it into your release. This is usually not a good idea, though.

Gigalixir prefers that you use environment variables for secrets and configuration. To do this, you'll want to delete your `prod.secret.exs` file, move the contents to your `prod.exs` file, and modify the values to pull from environment variables.

Open your `prod.exs` file and delete the following line if it is there

```
import_config "prod.secret.exs"
```

Then add the following in `prod.exs`

```
config :gigalixir_getting_started, GigalixirGettingStarted.Endpoint,
  http: [port: {:system, "PORT"}],
  server: true,
  secret_key_base: "${SECRET_KEY_BASE}"

config :gigalixir_getting_started, GigalixirGettingStarted.Repo,
  adapter: Ecto.Adapters.Postgres,
  url: {:system, "DATABASE_URL"},
  database: "",
  ssl: true,
  pool_size: 10
```

Replace `:gigalixir_getting_started` and `GigalixirGettingStarted` with your app name. You don't have to worry about setting your `SECRET_KEY_BASE` config because we generate one and set it for you. If you use a database, you'll have to set the `DATABASE_URL` yourself. You can do this by running the following. For more information on setting configs, see *How to Configure an App*.

Also, note that `server: true` is configured. That is required as well.

```
gigalixir set_config $APP_NAME DATABASE_URL "ecto://user:pass@host:port/db"
```

Optional Modifications

These modifications are not required, but are recommended if you want to use all of features Gigalixir offers. If you want to see the difference between `mix phoenix.new` and `gigalixir-getting-started` take a look at *the diff*.

Set up Node Clustering with Libcluster

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

Set Up Migrations

In development, you use `Mix` to run database migrations. In production, `Mix` is not available so you need a different approach. Instructions on how to set up and run migrations are described in more detail in *How to Run Migrations*.

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*.

TODO

I'm learning Elixir and need access to all Elixir's features

TODO

I'm using Heroku, but I've run into limitations

TODO - Save money too

I'm using AWS, but I'm spending too much time on infrastructure

TODO - Distillery - Kubernetes

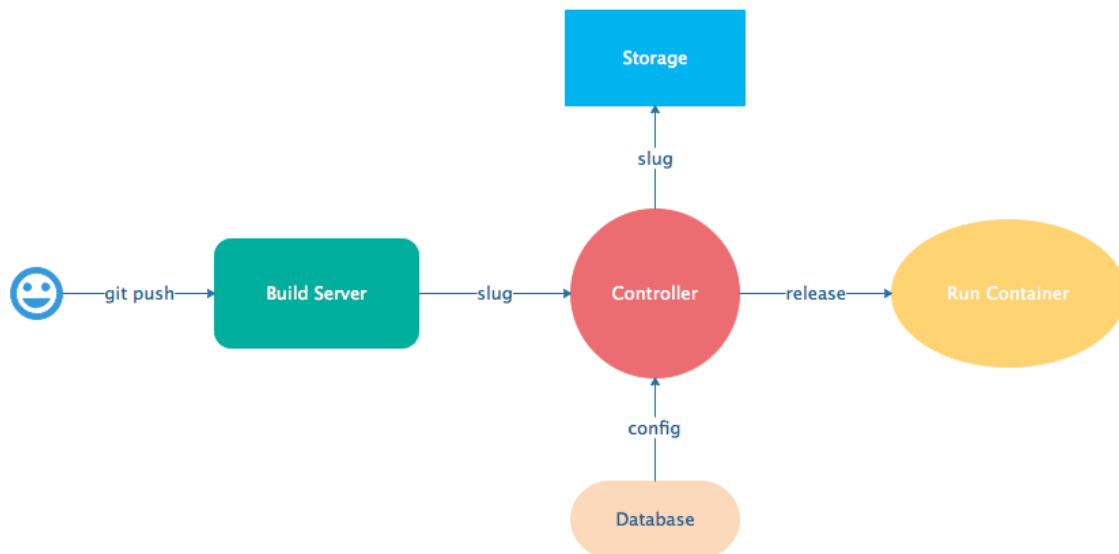
CHAPTER 5

Known Issues

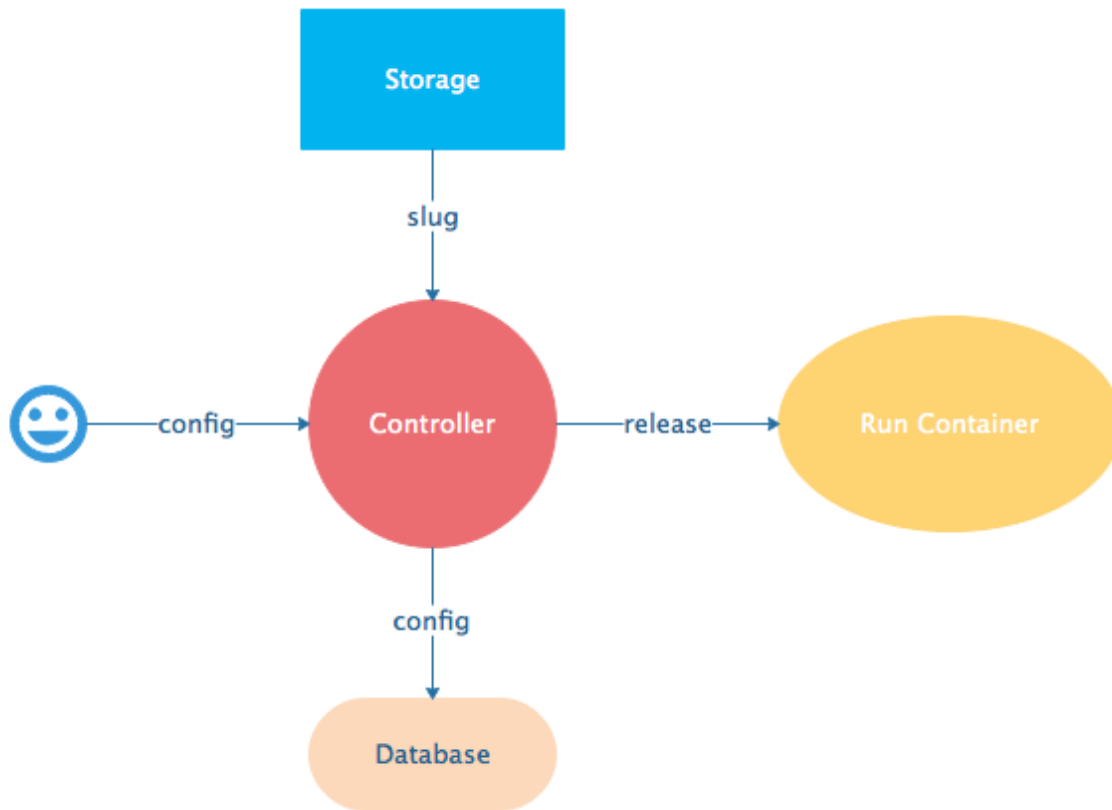
- Warning: Multiple default buildpacks reported the ability to handle this app. The first buildpack in the list below will be used.
 - This warning is safe to ignore. It is a temporary warning due to a workaround.
- curl: (56) GnuTLS recv error (-110): The TLS connection was non-properly terminated.
 - 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.

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.



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.

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 expires every 365 days.
- *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.

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 `.buildpack` file.

By default, the buildpacks we use include

- <https://github.com/gigalixir/gigalixir-buildpack-clean-cache.git>
 - To clean the cache if enabled.
- <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 phoenix.digest
- <https://github.com/gigalixir/gigalixir-buildpack-distillery.git>
 - To run mix release

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.

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.

Cleaning Your 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
```

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.

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](#).

Node and NPM versions are handled by the `heroku-buildpack-phoenix-static` buildpack. To configure, see the [heroku-buildpack-phoenix-static configuration](#).

Frequently Asked Questions

Do you support umbrella apps?

Yes! Just make sure you set `server: true` in your `prod.exs` and when you run migrations, use the `--migration_app_name` flag to specify which inner app has your migrations. Also, for static assets, be sure to set your `phoenix_relative_path`, see the [heroku-buildpack-phoenix-static](#) configuration. Also, if you set multiple Distillery releases in your `rel/config.exs` file, be sure your default release is set properly.

Do you support Phoenix 1.3?

Yes! Just be sure you set your `phoenix_relative_path`, see the [heroku-buildpack-phoenix-static](#) configuration.

Do you support Elixir 1.5 or OTP 20?

Yes! We support all versions of Elixir and OTP. See [How do I specify my Elixir, Erlang, Node, NPM, etc versions?](#).

Can I have multiple custom domains?

Yes! Just follow [How to Set Up a Custom Domain](#) for each domain.

What is Elixir? What is Phoenix?

This is probably best answered by taking a look at the [elixir homepage](#) and the [phoenix homepage](#).

How is Gigalixir different from Heroku and Deis Workflow?



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.

For a feature comparison table between Gigalixir and Heroku see, [Gigalixir vs Heroku Feature Comparison](#).

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. Heroku has a web interface, databases as a service, and tons of add-ons.

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.

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.

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 a Redis-backed queue to process jobs, take a look at Exq, but consider [whether you really need Exq](#).

What if Gigalixir shuts down?

Gigalixir was built as a labor of love. We want to see Elixir grow and this is our way of helping make that happen. Although making money is nice, that is not our primary goal.

My git push was rejected

Try force pushing with

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

Clustering Nodes

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"}
```

Then add `:elixir':libcluster'` and `:ssl` to your applications list. This may be different in Elixir 1.4. We'll update these instructions once we officially support Elixir 1.4. For a full example, see [gigalixir-getting-started's mix.exs file](#).

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: [
        kubernetes_selector: "${LIBCLUSTER_KUBERNETES_SELECTOR}",
        kubernetes_node_basename: "${LIBCLUSTER_KUBERNETES_NODE_BASENAME}"]]]
...

```

You also need to create a `rel/vm.args` file with something like this in it. For a full example, see [gigalixir-getting-started's vm.args file](#).

```
## Name of the node
-name ${MY_NODE_NAME}

## Cookie for distributed erlang
-setcookie ${MY_COOKIE}
...

```

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
...
```

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 10

Tiers

Gigalixir offers 2 tiers of pricing. The free tier is free, but you are limited to 1 size 0.2 instance and 1 size 0.6 database. The database is also limited to 10,000 rows.

Feature	FREE Tier	STANDARD Tier
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
Flexible Instance Sizes		YES
Horizontal Scaling		YES
Clustering		YES

CHAPTER 11

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
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 12

Pricing Details

In the free tier, everything is free. Once you upgrade to the standard tier, you pay \$50 per GB-month of memory. CPU, bandwidth, and power are included. You get 1 CPU share per GB of memory. See *Replica Sizing*.

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*.

- A replica is a docker container that your app runs in.
- Replica sizes are available in increments of 0.1 between 0.2 and 16.
- 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.

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 its 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>

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.

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.

CHAPTER 16

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](#).

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. Gigalixir automatically sets `REPLACE_OS_VARS=true` for you so all you have to do is add something like this to your `config.exs` file, set your app config, and you should be good to go. For information about how to set app configs, see [How to Configure an App](#).

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

Then set `MY_CONFIG`, by running

```
gigalixir set_config MY_CONFIG foo
```

In your app code,

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


CHAPTER 18

Troubleshooting

- ~/.netrc access too permissive: access permissions must restrict access to only the owner
 - `run chmod og-rwx ~/.netrc`

CHAPTER 19

Support/Help

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

The Gigalixir Command-Line Interface

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

Installation

Install `gigalixir` using `pip install gigalixir`. If you don't have `pip` installed, take a look at the [pip documentation](#).

Upgrade

To upgrade the Gigalixir CLI, run

```
pip install -U gigalixir
```

Encryption

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

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.

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.

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!

Open Source

The Gigalixir CLI is open source and we welcome pull requests. See [the gigalixir-cli repository](#).

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 22

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
```

How to Upgrade an Account

The free tier is limited to 1 instance and 1 database. If you need more, or you need to increase the size of either, you'll need to upgrade your account. To upgrade, first add a payment method

```
gigalixir set_payment_method
```

Then upgrade.

```
gigalixir upgrade
```


CHAPTER 24

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. There is currently no way to change your app name.

```
gigalixir create
```


CHAPTER 25

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 26

How to Deploy a Branch

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

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

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
```

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
  - git push gigalixir HEAD:master
```

For a full example, see [the gigalixir-getting-started repo](#).

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.

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 set_git_remote $APP_NAME
```

If you prefer to do it manually, run

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


CHAPTER 30

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 scale $APP_NAME --replicas=2 --size=0.6
```

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 does not automatically restart your app so you may need to do that yourself. We do this to give you more control at the cost of simplicity. It also potentially enables hot config updates or updating your environment variables without restarting. For more information on hot configuration, see [How to Hot Configure an App](#). For more information about using environment variables for app configuration, see [The Twelve-Factor App's Config Factor](#). For more information about using environment variables in your Elixir app, see [Using Environment Variables in your App](#).

```
gigalixir configs $APP_NAME
gigalixir set_config $APP_NAME FOO bar
gigalixir delete_config $APP_NAME FOO
```


CHAPTER 32

How to Hot Configure an App

This feature is still a work in progress.

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
```

How to Rollback an App

To rollback one release, run the following command.

```
gigalixir rollback $APP_NAME
```

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 $APP_NAME
```

You should see something like this

```
[
  {
    "created_at": "2017-04-12T17:43:28.000+00:00",
    "version": "5",
    "sha": "77f6c2952129ffecccc4e56ae6b27bbale65a1e3",
    "summary": "Set `DATABASE_URL` config var."
  },
  ...
]
```

Then specify the version when rolling back.

```
gigalixir rollback $APP_NAME --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.

How to Set Up a Custom Domain

Important! Right now, if you have multiple custom domains set for your app, all domain DNS entries must point to `tls.gigalixir.com`. Let's Encrypt issues a single certificate for all N domains on your app. If any single domain can not be authorized, the entire certificate fails to be provisioned. We're working on mitigating this issue.

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, first modify your DNS records and point your domain to `tls.gigalixir.com` using a CNAME record. Then, run the following command to add a custom domain.

```
gigalixir add_domain $APP_NAME www.example.com
```

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](#).

CHAPTER 36

How to Set Up SSL/TLS

SSL/TLS certificates are set up for you automatically assuming your custom domain is set up properly. You shouldn't have to lift a finger. For more information on how this works, see [How SSL/TLS Works](#).

CHAPTER 37

How to Tail Logs

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

```
gigalixir logs $APP_NAME
```

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 add_log_drain $APP_NAME $URL
# e.g. gigalixir add_log_drain $APP_NAME https://$TIMBER_API_KEY@logs.timber.io/frames
# e.g. gigalixir add_log_drain $APP_NAME syslog+tls://logs123.papertrailapp.com:12345
```

To show all your drains, run

```
gigalixir log_drains $APP_NAME
```

To delete a drain, run

```
gigalixir delete_log_drain $APP_NAME $DRAIN_ID
```


CHAPTER 39

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 add_ssh_key "ssh-rsa <REDACTED> foo@gigalixir.com"
```

To view your SSH keys

```
gigalixir ssh_keys
```

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

```
gigalixir delete_ssh_key 1
```

How to SSH into a Production Container

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 add_ssh_key "ssh-rsa <REDACTED> foo@gigalixir.com"
```

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 ssh $APP_NAME
```


CHAPTER 41

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 42

How to List Releases

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

```
gigalixir releases $APP_NAME
```

How to Change or Reset Your Password

To change your password, run

```
gigalixir change_password
```

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 send_reset_password_token
```


CHAPTER 44

How to Change Your Credit Card

To change your credit card, run

```
gigalixir set_payment_method
```


CHAPTER 45

How to Delete an App

There is currently no way to completely delete an app, but if you scale the replicas down to 0, you will not incur any charges. We are working on implementing this feature.

CHAPTER 46

How to Delete your Account

There is currently no way to completely delete an account. We are working on implementing this feature.

CHAPTER 47

How to Restart an App

Currently, restarts will cause brief downtime as we restart all containers at once. To avoid downtime, consider doing a hot upgrade instead. See, [How to Hot Upgrade an App](#). We are working on adding health checks so we can do rolling restarts with no downtime.

```
gigalixir restart $APP_NAME
```


CHAPTER 48

How to Run Jobs

There are many ways to run one-off jobs and tasks with Distillery. The approach described here uses Distillery's `command` command. As an alternative, you can also *drop into a remote console* and run code manually or use Distillery's custom commands, `eval` command, `rpc` command, pre-start hooks, and probably others.

To run one-off jobs, you'll need to write an Elixir function within your app somewhere, for example, `lib/tasks.ex` maybe. Gigalixir uses Distillery's `command` command to run your task.

```
gigalixir run $APP_NAME $MODULE $FUNCTION
```

For example, the following command will run the `Tasks.migrate/0` function.

```
gigalixir run myapp Elixir.Tasks migrate
```

The task is not run on the same node that your app is running in. We start a separate container to run the job so 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. Jobs are currently killed after 5 minutes.

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, *Tiers*.

CHAPTER 49

How to Reset your API Key

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

```
gigalixir reset_api_key
```

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

CHAPTER 50

How to Log Out

```
gigalixir logout
```


CHAPTER 51

How to Log In

```
gigalixir login
```

This modifies your `~/.netrc` file so that future API requests will be authenticated. API keys expire after 365 days, but if you login again, you will automatically receive an we API key.

How to provision a PostgreSQL database

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

```
gigalixir create_database $APP_NAME --size=0.6
```

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

```
gigalixir databases $APP_NAME
```

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](#).

If you are on the free tier, you will be limited to 10,000 rows. For information on upgrading your account, see [upgrade account](#).

How to scale a database

To change the size of your database, run

```
gigalixir scale_database $APP_NAME $DATABASE_ID --size=1.7
```

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

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 delete_database $APP_NAME $DATABASE_ID
```

How to install a Postgres Extension

First, make sure Google Cloud SQL supports your extension by checking [their list of extensions](#). If it is supported, [contact us](#) and we'll install it for you. We're working on automating this process, but in the meantime, we usually respond within an hour or so.

Database Sizes & Pricing

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. We currently do not set a limit for disk size, but we probably will later.

In the free tier, you get a size 0.6 database for free, but it is limited to 10,000 rows.

Size	Price / Month
0.6	FREE

Once you upgrade to the standard tier, the pricing is as follows.

Size	Price / Month
0.6	\$25
1.7	\$50
4	\$400
8	\$800
16	\$1600
32	\$3200
64	\$6400
128	\$12800

Prices are prorated to the second.

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: 10
```

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 PostgreSQL database*, then `DATABASE_URL` should be set for you automatically once the database is provisioned. Otherwise,

```
gigalixir set_config $APP_NAME 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 PostgreSQL database*. If you prefer to provision your database manually, follow *How to set up a Google Cloud SQL PostgreSQL database*.

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 publically 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 set_config $APP_NAME DATABASE_URL "ecto://postgres:$PASSWORD@$EXTERNAL_  
↪IP:5432/$DB_NAME"
```

with \$APP_NAME, \$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 box so 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.

How to Run Migrations

In order to run migrations, 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 add_ssh_key "ssh-rsa <REDACTED> foo@gigalixir.com"
```

We provide a special command to run migrations.

```
gigalixir migrate $APP_NAME
```

Since Mix is not available in production with Distillery, 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*.

Also note that because we don't spin up an entire new node just to run your migrations, migrations are free. Also, this doesn't yet work if you have an umbrella app and the app the migrations are in is a different name from your release name.

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 migrate $APP_NAME --migration_app_name=$MIGRATION_APP_NAME
```

If you need to tweak the migration command, all we are 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, 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 mix release --env=prod  
MIX_ENV=prod DATABASE_URL="$YOUR_PRODUCTION_DATABASE_URL" mix ecto.migrate
```

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.

```
Path.join([priv_dir(:my_app), "repo", "seeds.exs"])  
Code.eval_file(seed_script)
```


CHAPTER 60

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 add_ssh_key "ssh-rsa <REDACTED> foo@gigalixir.com"
```

Then run this command to drop into a remote console.

```
gigalixir remote_console $APP_NAME
```

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 distillery $APP_NAME $COMMAND
```


CHAPTER 62

How to Check App Status

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

```
gigalixir status $APP_NAME
```

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
```

How to Launch a Remote Observer

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 add_ssh_key "ssh-rsa <REDACTED> foo@gigalixir.com"
```

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 adds it by default, but you have to add it yourself in Phoenix 1.2.

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

```
gigalixir observer $APP_NAME
```

and follow the instructions. 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.

How to see the current period's usage

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

```
gigalixir current_period_usage
```

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

CHAPTER 66

How to see previous invoices

To see all your previous period's invoices, run

```
gigalixir invoices
```

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 uses a 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 are TLS encrypted.

CHAPTER 68

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 69

Indices and Tables

- `genindex`
- `modindex`
- `search`