GIGALIXIR Documentation

Release 0.11.0

Invisible Software, Inc.

Jun 20, 2017

Contents:

GIGALIXIR is a platform-as-a-service designed just for Elixir and Phoenix.

CHAPTER 1

What is GIGALIXIR?

GIGALIXIR is a Platform-as-a-Service designed for Elixir and Phoenix apps. PaaSes are designed to make it simple to deploy, run, and manage an app in production, but GIGALIXIR is unique because it is designed to support all of features that probably drew you to Elixir in the first place, like node clustering, hot upgrades, and remote observer. Moreover, some platforms restart your processes every 24 hours, restrict you to 50 simultaneous connections per instance, and limit each connection to 30 seconds. GIGALIXIR made opinionated design decisions from the ground up to support all the fetaures of Elixir because we know that most Elixir apps are built to be distributed, highly available, and handle large numbers of concurrent long-lived connections.

CHAPTER 2

Quick Start

Prequisites

1. Make sure you have `pip` installed. For help, take a look at the pip documentation.
2. Make sure you have `python2.7`, not `python3`.
3. Make sure you are on Linux or OS X. Windows users have been using a small Linux instance in the cloud to use GIGALIXIR.
4. Make sure you have a beta invitation. If you don't have one, request one using the beta sign up form.
5. Elixir 1.3 is officially supported. Elixir 1.4 is known to work, but a lot of the documentation assumes you are on 1.3. We are working on officially supporting 1.3. You can configure the production version in your buildpack configuration file.
6. Phoenis 1.2 is officially supported. Phoenix 1.3 is known to work, but a lot of the documentation assumes you are on 1.2. We are working on officially supporting 1.3. For a working example of Phoenix 1.3 and Elixir 1.4, see gigalixir-getting-started-phx-1-3-rc-2.
7. Umbrella apps are known to work, but they are not yet officially supported. We are working on adding official support.

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
```

Create 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. It will also prompt you for credit card information. GIGALIXIR currently does not offer a free trial, but we do offer a *money back guarantee*. Please don't hesitate to use it.

```
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
```

Prepare Your App

There are a few steps involved to *make your existing app work on GIGALIXIR*, but if you are starting a project from scratch, we recommend you 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 so you can later run `git push gigalixir master`. 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)
```

Deploy!

Finally, build and deploy.

```
git push gigalixir master
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
```

Verify logs

Make sure your logs look good

```
gigalixir logs $APP_NAME
```

What's Next?

- *How to Hot Upgrade an App*
- *How to Configure an App*
- *How to Scale an App*

CHAPTER 3

Modifying an Existing App to Run on GIGALIXIR

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,
  server: true,
  secret_key_base: "${SECRET_KEY_BASE}"

config :gigalixir_getting_started, GigalixirGettingStarted.Repo,
  adapter: Ecto.Adapters.Postgres,
  url: {:system, "DATABASE_URL"},
  ssl: true,
  pool_size: 20
```

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 running hot upgrades, see *How to Hot Upgrade an App* and *Life of a Hot Upgrade*.

CHAPTER 4

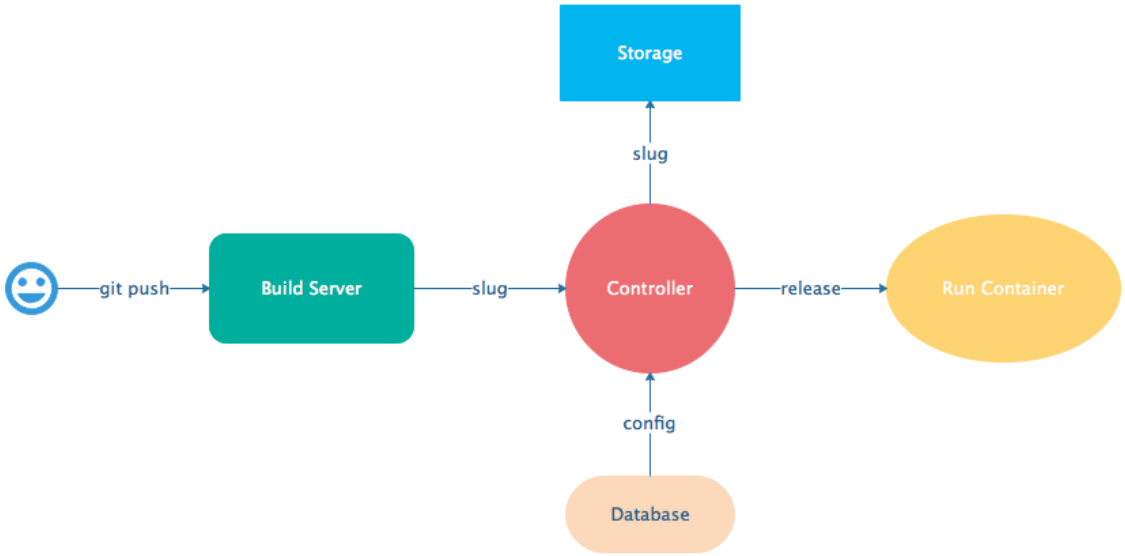
Known Issues

- 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.

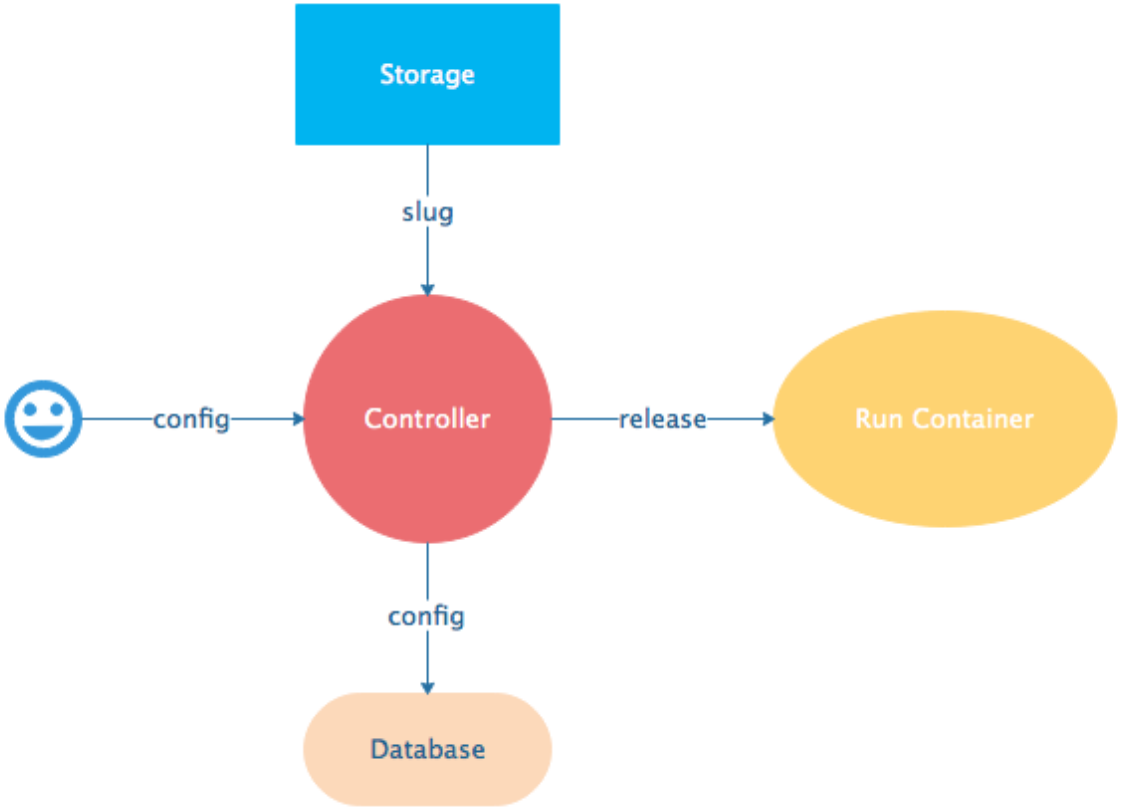
CHAPTER 5

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 usethem 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.

```
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.

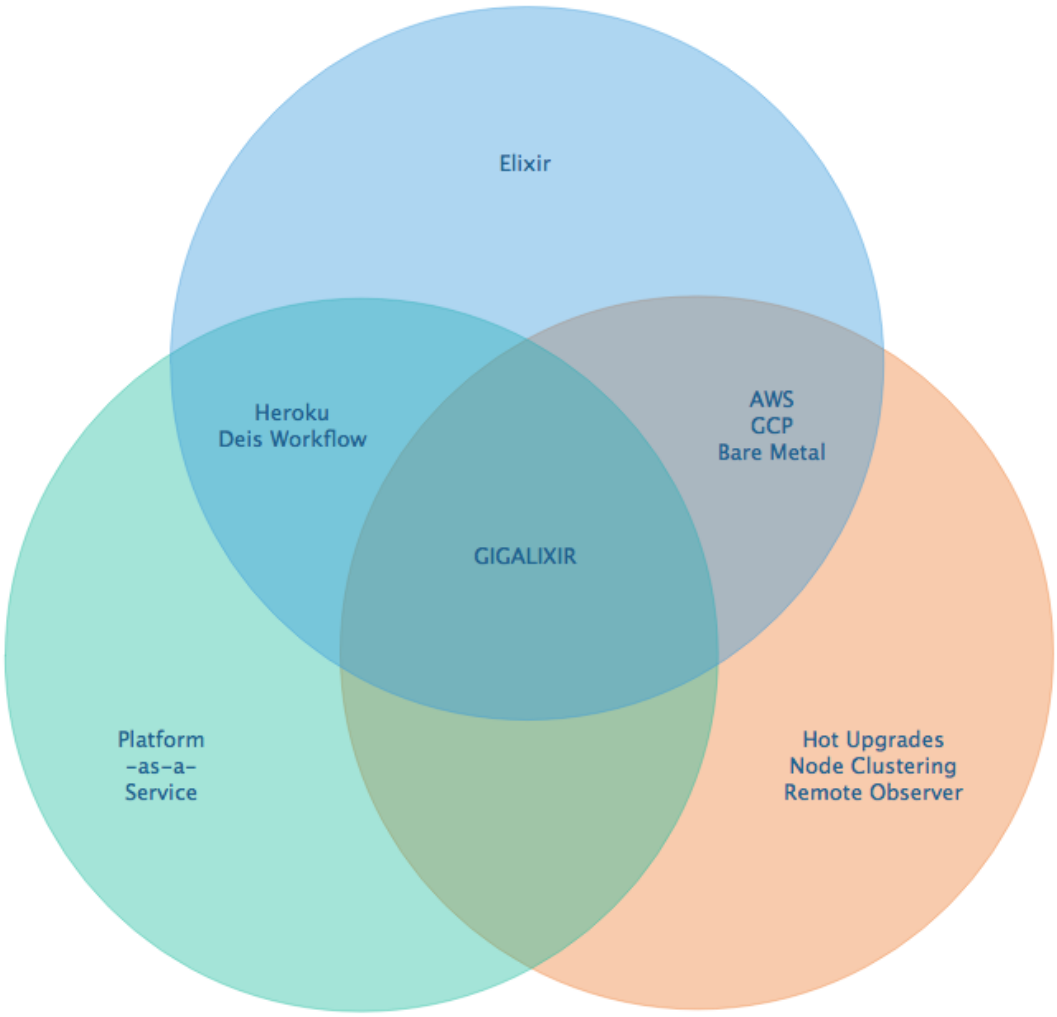
CHAPTER 6

Frequently Asked Questions

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 to 50 per instance. 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. 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 an 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
```

CHAPTER 7

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 8

Pricing Details

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.
```

CHAPTER 9

Replica Sizing

- A replica is a docker container that your app runs in.
- Replica sizes are available in increments of 0.1 between 0.5 and 16. Contact us if you need a bigger size.
- 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 10

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 11

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.

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 12

Monitoring

GIGALIXIR doesn't provide any monitoring out of the box, but we are working on it.

CHAPTER 13

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 14

Troubleshooting

TODO: Common issues go here.

CHAPTER 15

Support/Help

If you run into issues, Stack Overflow is the best place to search. If you can't find an answer, the developers at GIGALIXIR monitor the gigalixir tag and will answer questions there. We prefer Stack Overflow over a knowledge base because it is public and collaborative. If you have a private question, email help@gigalixir.com or call us at (415) 326-8880. With GIGALIXIR, you always get support from developers, not customer support representatives. We are very responsive and we are available 24/7. If we become too big, it's possible we won't be able to offer this level of support one day, but we think it is extra important for a startup to provide above-and-beyond support.

CHAPTER 16

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.

CHAPTER 17

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 18

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. It will also prompt you for credit card information. GIGALIXIR currently does not offer a free trial, but we do offer a *money back guarantee*. Please don't hesitate to use it.

```
gigalixir signup
```

CHAPTER 19

How to Create an App

To create your app, run the following command. It will also set up a git remote so you can later run `git push gigalixir master`. 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 20

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 21

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
```

CHAPTER 22

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
```

CHAPTER 23

How to Configure an App

All app configuration is done through envirnoment 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 configs $APP_NAME
{
  "FOO": "bar"
}
$ gigalixir delete_config $APP_NAME FOO
$ gigalixir configs $APP_NAME
{}
```

CHAPTER 24

How to Hot Configure an App

This feature is still a work in progress.

CHAPTER 25

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 26

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 foo
[
  {
    "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 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.

CHAPTER 27

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, 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 28

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 29

How to Tail Logs

You can tail logs in real-time aggregated across all containers using the following command. Note that it takes up to a minute or so to start streaming logs because it sets up a Stackdriver sink and PubSub topic on-demand. We're working on improving this, but if you need more logging features, we suggest PaperTrail. We have tested and verified that it works.

```
gigalixir logs $APP_NAME
```

CHAPTER 30

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
```

CHAPTER 31

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 32

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 33

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
```

CHAPTER 34

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 35

How to Change Your Credit Card

To change your credit card, run

```
gigalixir set_payment_method
```

CHAPTER 36

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 37

How to Delete your Account

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

CHAPTER 38

How to View Billing and Usage

We currently do not have a way to view usage or your bill so far in the middle of the month, but we are working on it. For more information about how your bill is calculated, see *Pricing Details*.

CHAPTER 39

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 40

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 foo
```

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, *Pricing Details*.

CHAPTER 41

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 42

How to Log Out

```
gigalixir logout
```

CHAPTER 43

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.

CHAPTER 44

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.

CHAPTER 45

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*.

CHAPTER 46

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
```

CHAPTER 47

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

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.

CHAPTER 48

How to Connect a Database

If you followed the quick start, 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"},
  ssl: true,
  pool_size: 20
```

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 in 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 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 49

How to Run Migrations

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.

CHAPTER 50

How to Drop into a Remote Console

```
gigalixir remote_console $APP_NAME
```

CHAPTER 51

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 52

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
```

CHAPTER 53

How to Launch a Remote Observer

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 youself 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.

CHAPTER 54

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 55

How to see previous invoices

To see all your previous period's invoices, run

```
gigalixir invoices
```

CHAPTER 56

How to check my credit balance

If you have a credit balance on your account, you can see it by running

```
gigalixir credit
```

CHAPTER 57

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 58

Indices and Tables

- genindex
- modindex
- search