Deploying a Phoenix application that uses SQLite to Gigalixir.

Caleb Josue Ruiz Torres.

November 2, 2023.

Prerequisites: You have installed the necessary Software required to write Phoenix applications; you are also able to write a "Hello world" Phoenix application at least. This guide assumes you are using Linux. (You can use it on Windows by means of the WSL2 facility). If you are just getting started, don't worry. Installation instructions for Erlang, Elixir, and Phoenix, can be found here. Of course, you will also need to install SQLite. And git! Did I mention you have to have installed Gigalixir CLI? Yes, that's right. Install it!

Introduction.

Gigalixir categorizes itself as a Platform as a Service (PAAS). Supporting applications written in Elixir and Phoenix.

As a Software Developer working in the implementation of an idea, you'd like to concentrate on the features you are delivering to your customers instead of taking care of the nuances related to the configuration arriving with an Infrastructure as A Service offering.

This document will explain the steps required to deploy an application to Gigalixir, we will be creating a new application, but if you already have an existing one, you are covered too in the documentation, by the way Gigalixir characterizes itself as having great documentation and great customer support.

At the time of writing this document, no credit card was required to create an account in their web portal. I really consider these guys to be in the business of enabling people to deliver their ideas to the public.

Let's get started.

Gigalixir Account Creation and Software Installation.

I am not saying a lot here, go ahead and create an account in Gigalixir to be able to publish your application.

As mentioned in the prerequisites, I assume you have installed the necessary Software to write Phoenix applications on your development machine.

Now, in order to deploy to Gigalixir, You will also need to install NVM. What's NVM? Don't panic, it is just a matter of executing a single command. If you did it well, you should be able to check for the NVM version using `nvm --version` in your terminal.

Let's use **nvm** to install Node, execute `nvm install v21.1.0`

If you have several versions for Erlang and Elixir, I suppose you know have set them respectively as active. If you just have one version installed, cool. Let's set the Node version we have just installed as active with `nvm use 21.1.0`

Great.

Let's move ahead and illustrate the deployment process.

The creation of the local application to be deployed.

Move to the directory you'd like to use to place your application. I am using playground.

```
cd playground
```

Time to create the application we want to deploy to Gigalixir. Let's use the following command to accomplish this task.

```
mix phx.new <YourAppName> --install --database sqlite3
```

The install flag is there so mix can take care of the dependencies from the very beginning and not asking about it while creating the application. Remember, we are using SQlite, so we need to inform about it by means of the database flag.

Let's switch to our recently created application directory.

```
cd <YourNameApp>
```

If you list the directory content, you will see there is no file supporting the database. We need to create it.

```
mix ecto.create
```

Try again listing the files present in this directory and now you'll find the ones ending with .db

Let's see if our application works locally!

```
mix phx.server
```

Outstanding. Deploying to Gigalixir is a matter of making a push of our code to the repository living in their servers (We will create it later). For now, let's transform our current application directory into a git repository. Assisted by the following commands.

```
git init
```

```
git add .
```

```
git commit -m "master: Initial commit"
```

Now, we are using the branch *master* here but in a large project make sure to use some methodology like GitFlow.

We need to create some extra files in order to help Gigalixir do its magic. Since most likely we will be creating tons of applications, I decided to store those simple commands in a file.

Create **a gigalixir_build_setup.sh** in your home directory and copy paste the following content into it.

```
echo "elixir_version=1.15.4" > elixir_buildpack.config

echo "erlang_version=26.0.2" >> elixir_buildpack.config

echo "node_version=21.1.0" > phoenix_static_buildpack.config

echo '{

    "scripts": {

        "deploy": "cd .. && mix assets.deploy && rm -f _build/esbuild"

    }

}' > assets/package.json

echo "# this file is here so Gigalixir uses Elixir Releases on deploy" >
config/releases.exs

echo '...done'
```

Make sure you change the versions depicted here so you can reflect what's installed on your machine instead. Also, change the file so it is executable, using the next command.

```
chmod +x gigalixir_build_setup.sh
```

We can now execute the script.

```
~/gigalixir_build_setup
```

The above command will create three new files for us. Let's commit the recently added change.

```
git add .
```

```
git commit -m "master: Gigalixir build setup"
```

Creating the Gigalixir Application.

We are going to create the Gigalixir application, that is, the application that is going to be live and ready for your audience to interact with. You can use whatever version of the command is most suitable for you. As portrayed in the official documentation we are also storing the application name for later use.

Option 1.

```
APP_NAME=$(gigalixir create)
```

Option 2.

```
APP_NAME=$(gigalixir create --name <ChooseYourAppName>)
```

The environment variable APP_NAME now holds the value for your application name, which by the way, the option 1 assigns a random name for you, so you don't have to think about it. We are about to use to set up the web endpoint.

```
gigalixir config:set PHX_HOST=${APP_NAME}.gigalixirapp.com
```

```
gigalixir config:set PHX_SERVER=true
```

We need to update the path for the database in a couple of places, the first happens to be in the file **config/dev.exs**

Find the line in which the database path is specified in this file and change the path to point to your database file, this line is around the beginning of this file. Most likely this change does not need to be made. but still, make sure it points to the right direction.

```
database: Path.expand("../yourappname_dev.db", Path.dirname(__ENV__.file)),
```

It is now the turn for the **config/runtime.exs** file to be updated.

Around line of code 24, we need to update the database path, the same as before.

```
Path.expand("../yourappname_dev.db", Path.dirname(__ENV__.file))
```

Set the host alternative to point to your application instead of example.com in line of code 47 (Look for the host entry).

```
host = System.get_env("PHX_HOST") || "emptyapp.gigalixirapp.com"
```

Let's test the application to see if it is working locally, run the following command in your terminal.

```
mix phx.server
```

Commit your recent changes to the repository.

```
git add config/runtime.exs
```

```
git commit -m "master: Database path has been updated"
```

At last, deploy the app with

```
git push -u gigalixir master
```

Make sure you specify the branch name you are using in this step, I have been using the default one until this point.

See the results in the console, and let's dive into the provided URL to see our application live.

We can stop here, since this is a document all about deploying a Phoenix application that uses SQLite to Gigalixir. But in all honesty, we haven't seen if the database interaction is really working. So, let's do it quickly by means of adding an authentication system, don't worry, it's all about integrating the one provided by Phoenix itself.

Authentication System

You'll be surprised by how easy this part actually is, of course, in production you'll need to take a lot of stuff into consideration, like the algorithms to encrypt the passwords and some other configuration recommendations so you can have a stronger application. We are using the authentication system provided by phoenix "as-is". Execute the following command.

```
mix phx.gen.auth Accounts User users
```

You will be asked what kind of mechanism you want to use; we are choosing Phoenix LiveView.

The instruction we have just executed added some new dependencies we need to bring in with

```
mix deps.get
```

Time for our first migration.

```
mix ecto.migrate
```

Make sure everything is running smoothly.

```
mix phx.server
```

And commit the changes to the repository.

```
git add .
```

```
git commit -m "master: Authentication system in place"
```

I'd like to disable the email notification system (Yes, Phoenix also provides one) before pushing our changes live again, because the topic is out of scope given the nature of the present document.

Direct yourself to the **lib/YourAppName/accounts/user_notifier.ex** file and comment the lines of code related to the email notification. Here they are. The next page depicts the method in its entirety.

```
# alias Emptyapp.Mailer - This is the line number 4 at the time of writing
# with {:ok, _metadata} <- Mailer.deliver(email) do
#   {:ok, email}
# end
{:ok, email}
```

As you can see, this last line of code is new, you should add it to the body of the private method *deliver*, and comment the ones also mentioned above. Around Lines of code 15, 16, 17.

Hmmm… You know what? Here's the method in its entirety so you can see its final version.

```
# lib/YourAppName/accounts/user_notifier.ex

#Delivers the email using the application mailer.
defp deliver(recipient, subject, body) do
    email =
        new()
        |> to(recipient)
        |> from({"Emptyapp", "contact@example.com"})
        |> subject(subject)
        |> text_body(body)

    # with {:ok, _metadata} <- Mailer.deliver(email) do
    #  {:ok, email}
    # end
    {:ok, email}
end
```

Commit the change we have made, wait! Make sure your application continues to work locally.

```
git add lib/YourAppName/accounts/user_notfier.ex
```

```
git commit -m "master: Disabling email notifications"
```

Let's deploy our shining Authentication System.

```
git push
```

Now listen, your application is live. But how do you tell the server there are new tables to consider? We need to perform a migration. Even better, we need a way to interact with the server. Let's create a **ssh key** for this purpose.

```
ssh-keygen -t ed25519 -C "yourEmail@example.com" -f ~/.ssh/filename
```

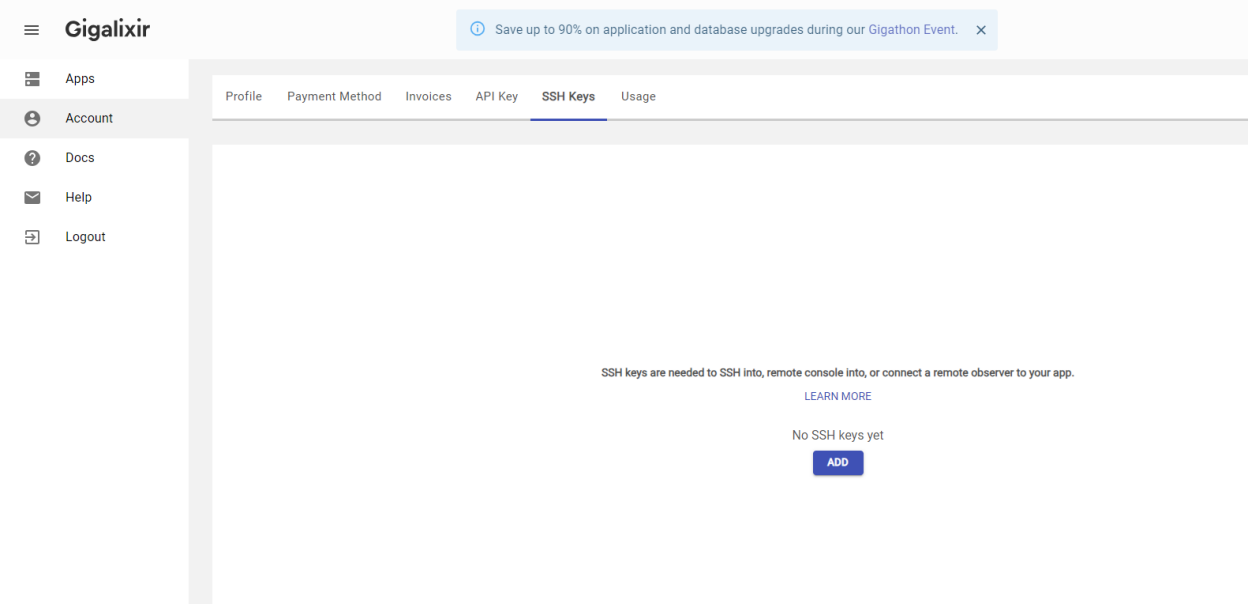
Enter the paraphrase you'd like to use; I left the field blank (Press enter).

Grasp (Copy) the content of your public key

```
cat .ssh/filename.pub
```

Log in to gigalixir.com and find the place in which you need to paste the content you just copied.

It looks like this.



Click **add** and paste the content of your public key.

That's it! Time to perform the migration.

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

Go to your live application and have fun while playing around with it.

Now, for real. Our fellows at Gigalixir did an amazing job documenting the whole stuff, make you sure you visit https://www.gigalixir.com/docs/ if you want to learn more about the possibilities you can take advantage of.