Deploying a Back-End Application with Render

Learn how to deploy a back-end application using Render!

Introduction

In this tutorial, we will practice using Render to deploy a back-end application. You may have experience with deploying a Node and Express application, but this application will also include a database and HTTP requests to an external API. Our app was built using Node and Express with endpoints for inserting data into a PostgreSQL database.

- **Forking the Back-End Application Code Repository**: fork a sample back-end application to use for our deployment
- **Configuring and Deployment with Render**: configure a Web Service Render deployment, including more advanced settings like configuring environment variables
- **Connecting to an Existing Database using Environment Variables**: connect an existing PostgreSQL database to our forked application source code
- **Verifying Deployment with Added Environment Variables**: verify a successful deployment after adding environment variables through Render

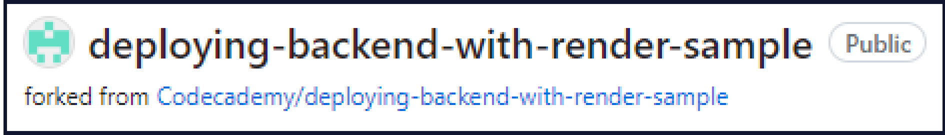
Note: This tutorial requires access to a GitHub account and assumes familiarity with using GitHub. If you are unfamiliar with GitHub or need help setting up an account, we recommend visiting the Learn GitHub: Introduction course first!
Let’s get started!

Forking the Back-End Application Code Repository

To get started with deploying a back-end application with Render, we will use an existing Node and Express back-end application that inserts randomly generated text of activity names into a database. The back-end also makes requests to an external API to retrieve text with randomized activity names. The application offers its own API endpoint that when accessed, calls the function that retrieves the random activity, and then inserts it into the database. A PostgreSQL database is used to keep track of these inserted activities.

For this tutorial, we will be using the `deploying-backend-with-render-sample` back-end application. We'll need to connect our GitHub account to Render and then connect our desired repository. For a refresher on the set-up process and how to fork a GitHub repository, check out Deploying a Simple App with Render.

After the repository has successfully been forked, we can confirm that the code now lives under our GitHub username ownership and was forked from `Codecademy`.


deploying-backend-with-render-sample Public

forked from Codecademy/deploying-backend-with-render-sample

Take a moment to browse the files in the repository closely.

Notice that the repository has a file called `app.js`, which holds the code used to retrieve the generated activity names and also creates the API endpoints to insert and retrieve the activities from our database.

There is also a `package.json` file which will later tell Render which Node packages need to be installed in order to build, deploy, and run our back-end application.

Now that our application code is ready and available within our repository, we can practice deploying it using Render.

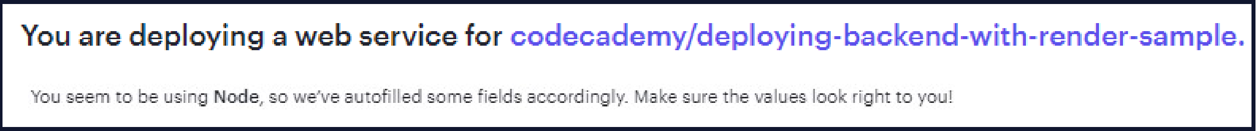
Deployment with Render

To start the deployment process, we need to navigate to the Render dashboard. Make sure to be logged in to see the dashboard. From the top-right of the menu, we can click the blue, "New +" button to create and configure our deployment. From the dropdown, select the "Web Service" option to deploy our application to a web server. For a recapon how to deploy a Web Service in Render, check out Deploying a Simple App with Render

We will then be able to connect the GitHub repository that we just forked in order to deploy it as a web service.

Configuring a Web Service

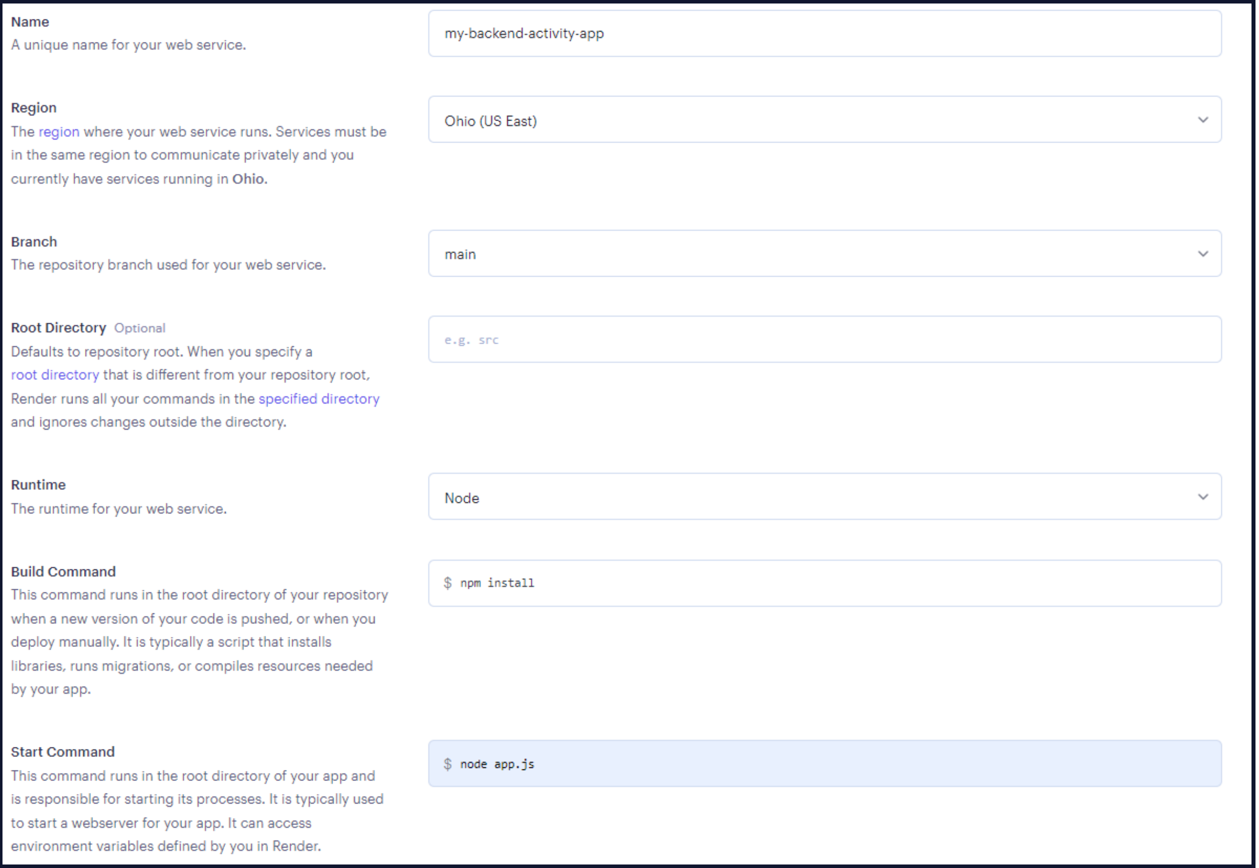
Notice that Render has automatically detected that we are using Node for our application and has pre-populated some of the configuration fields for us. As we step through the deployment settings, we can also modify any of the pre-populated fields as needed.



You are deploying a web service for codecademy/deploying-backend-with-render-sample.

You seem to be using Node, so we've autofilled some fields accordingly. Make sure the values look right to you!

Configure the **Name**, **Region**, and **Branch** settings for the deployment based on personal preferences and forked repository setup. Next, we will leave the **Root Directory** blank so that we use the repository's default root directory. We should see that the **Runtime** has been set to `Node` already, which we will leave as is since our application uses Node. For our **Build Command**, recall from viewing our forked source code files that we have a `package.json` file that lists which packages need to be installed in order to build and deploy our application. For this field, you may type `npm install` which will instruct Render to run that command during the deployment to install the necessary packages and dependencies. For the **Start Command** field, we can use the command `node app.js` to run our application.



Name A unique name for your web service.	my-backend-activity-app
Region The region where your web service runs. Services must be in the same region to communicate privately and you currently have services running in **Ohio**.	Ohio (US East)
Branch The repository branch used for your web service.	main
Root Directory Optional Defaults to repository root. When you specify a root directory that is different from your repository root, Render runs all your commands in the specified directory and ignores changes outside the directory.	e.g. src
Runtime The runtime for your web service.	Node
Build Command This command runs in the root directory of your repository when a new version of your code is pushed, or when you deploy manually. It is typically a script that installs libraries, runs migrations, or compiles resources needed by your app.	$ npm install
Start Command This command runs in the root directory of your app and is responsible for starting its processes. It is typically used to start a webserver for your app. It can access environment variables defined by you in Render.	$ node app.js

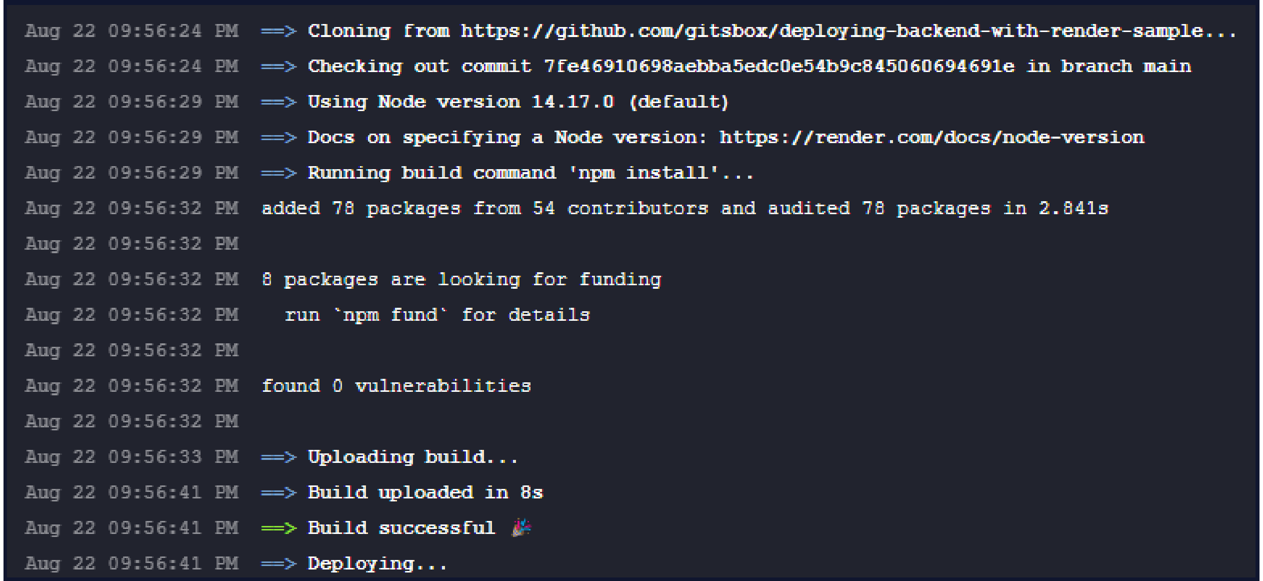
We can leave the "Instance Type" set to the "Free" tier.

We will discuss the "Advanced" menu settings in more detail later. For now, let's deploy our application as is!

Building and Deploying the Web Service

At the bottom of the web service configuration page, click on the "Create Web Service" button to start building and deploying the application.

We will be redirected to the log console window that shows the current build and install steps occurring to deploy the application. At first, the console will be empty, but as the deployment processes, we'll see our build and install commands being run:



```
Aug 22 09:56:24 PM  ==> Cloning from https://github.com/gitsbox/deploying-backend-with-render-sample...
Aug 22 09:56:24 PM  ==> Checking out commit 7fe46910698aebba5edc0e54b9c845060694691e in branch main
Aug 22 09:56:29 PM  ==> Using Node version 14.17.0 (default)
Aug 22 09:56:29 PM  ==> Docs on specifying a Node version: https://render.com/docs/node-version
Aug 22 09:56:29 PM  ==> Running build command 'npm install'...
Aug 22 09:56:32 PM  added 78 packages from 54 contributors and audited 78 packages in 2.841s
Aug 22 09:56:32 PM
Aug 22 09:56:32 PM  8 packages are looking for funding
Aug 22 09:56:32 PM    run `npm fund` for details
Aug 22 09:56:32 PM
Aug 22 09:56:32 PM  found 0 vulnerabilities
Aug 22 09:56:32 PM
Aug 22 09:56:33 PM  ==> Uploading build...
Aug 22 09:56:41 PM  ==> Build uploaded in 8s
Aug 22 09:56:41 PM  ==> Build successful 🎉
Aug 22 09:56:41 PM  ==> Deploying...
```

Note: While Render's default Node version is currently Node 14, we can update our Node version by following these steps to specify a Node version.
Once the deployment finishes, we should see a green "Live" state above the console window. Now, we can try accessing the application using the Render-provided URL at the top left. Clicking the provided link should open a new browser tab where our back-end application should be running. But wait! We will see the following error displayed instead:

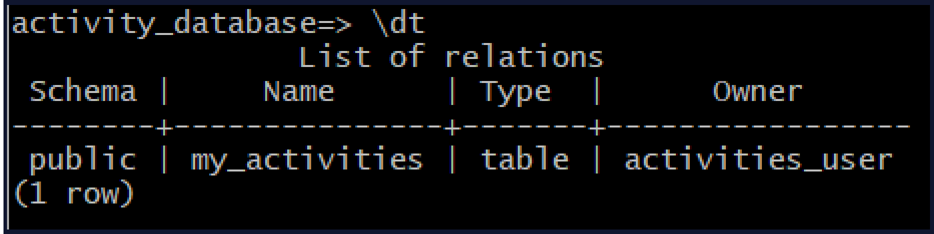
```
{"status":"error","message":"connect ECONNREFUSED 127.0.0.1:5432"}
```

The error indicates an issue connecting to the database. The default port number that PostgreSQL uses is 5432. So, the best place to start debugging is our database connection.

Returning to our code repository, in the `app.js` file, we can see that we attempt to use something called an **environment variable** to set the URL of our database. The environment variable we are trying to read from is called `DATABASE_URL`, but remember, we did not set any values for this. In fact, we don't even have a database to point our code to yet!

Fortunately, Render provides the ability to create a PostgreSQL database that we can access. To learn more about how to set up this PostgreSQL database in Render, or how to add tables to your database, take a look at the article on Creating a PostgreSQL Database with Render. Once you have your Render-created PostgreSQL database ready, take note of the "Internal Database URL" value from the "Connections" section as we will need it later.

Looking at our `app.js` file again, we can see that we are making database query calls to a table named `my_activities`. So be sure there is a PostgreSQL database that has at least one table called `my_activities` and that it contains one column called `activity`, which is a text data type. This table should resemble the following:



```
activity_database=> \dt
               List of relations
 Schema |     Name      | Type  |      Owner
--------+---------------+-------+-----------------
 public | my_activities | table | activities_user
(1 row)
```

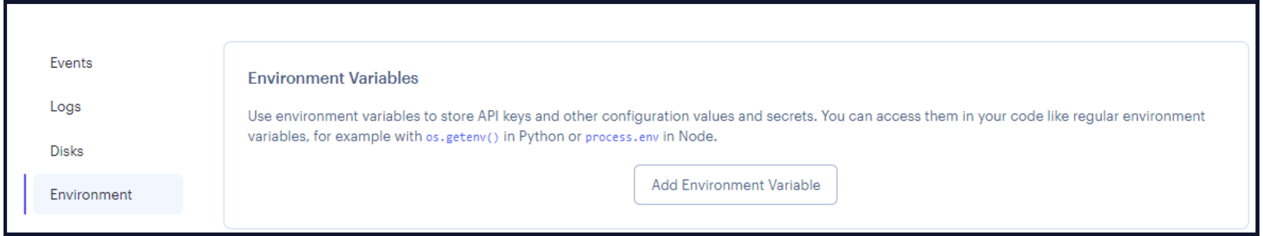
With a configured PostgreSQL database and table ready to go, we will learn how to connect to our database through our code using environment variables.

Connecting to an Existing Database using Environment Variables

We now have our active database instance that contains the table we need for our application. It's time to connect to it by adding environment variables. **Environment variables** are dynamic key and value pairs where the values can be updated or changed during the runtime of our code, that can affect the behavior of our applications. Recall, in our `app.js` file, we can see that we are trying to access an environment variable called `DATABASE_URL` that should have some associated value that equals the database URL string. But we can imagine that if we need to test our application, we can set up a dedicated database for testing — without affecting our production database. With our environment variable, we can set the `DATABASE_URL` to the exact database URL we need. This helps us separate our environments and reduce repetitive code.

Now that we've covered what environment variables are and how they can be used, let's navigate back to Render and access our deployed web service dashboard. We can do this by clicking "Dashboard" in the top menu and then selecting our `my-backend-activity-app` web service.

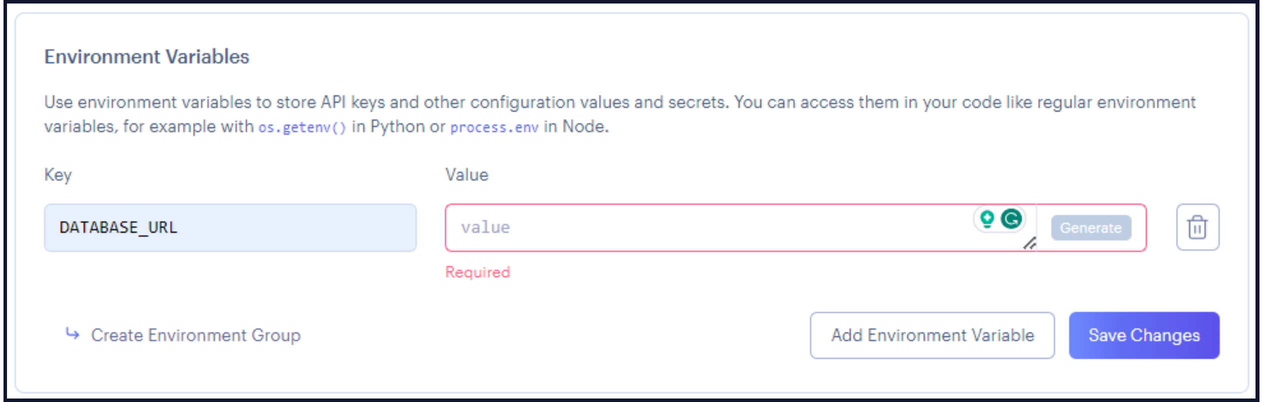
From the left-hand menu, we'll select "Environment". Then, we'll see a button to "Add Environment Variable". We'll click this to start adding the environment variable to link our internal database URL.



Environment Variables

Use environment variables to store API keys and other configuration values and secrets. You can access them in your code like regular environment variables, for example with `os.getenv()` in Python or `process.env` in Node.

For the "Key" field, supply the `DATABASE_URL` name that our `app.js` source code is searching for. For the "Value" field, supply the internal database URL from the PostgreSQL "Connections" information that we noted earlier. Note that we can also click the "Generate" button within the "Value" field in order to generate a random 32-character alpha-numeric value. This is helpful when we need to generate a random value for things like secret keys, passwords, and other confidential values.



Environment Variables

Use environment variables to store API keys and other configuration values and secrets. You can access them in your code like regular environment variables, for example with `os.getenv()` in Python or `process.env` in Node.

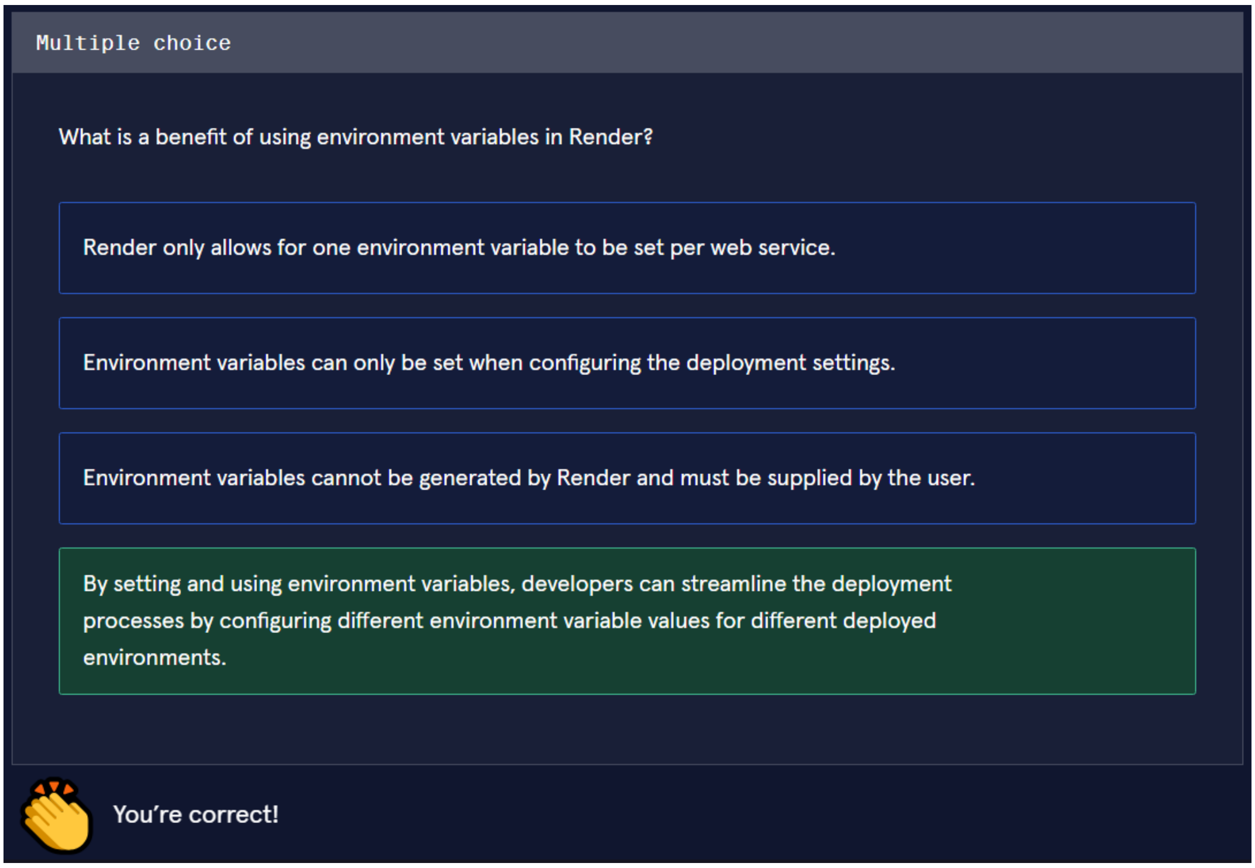
Key	Value
DATABASE_URL	value

Required

Notice that our `app.js` file also attempts to access an environment variable named `PORT` which should represent the port that the server serving the web service is running on.

Render defaults to running the web service on port 10000 so we will add another environment variable to set the port to 10000. We will click the "Add Environment Variable" button to add an additional environment variable with a **Key** set to `PORT` and **Value** set to `10000`.

Click the "Save Changes" button to save the environment variable.



Multiple choice

What is a benefit of using environment variables in Render?

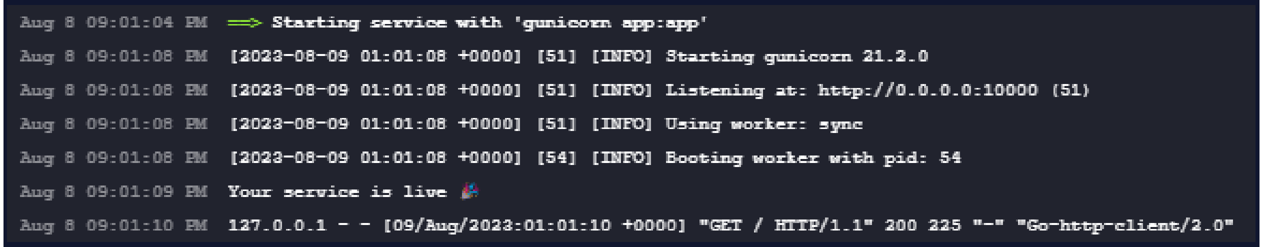
- Render only allows for one environment variable to be set per web service.
- Environment variables can only be set when configuring the deployment settings.
- Environment variables cannot be generated by Render and must be supplied by the user.
- [✓ By setting and using environment variables, developers can streamline the deployment processes by configuring different environment variable values for different deployed environments.]

You're correct!

Verifying Deployment with Added Environment Variables

When an environment variable is added or modified, Render will automatically redeploy the web service.

In our deploy console window, we should see the application build and deploy successfully, and we will see the green "Live" indicator at the top again once the deployment is complete.



```
Aug 8 09:01:04 PM  ==> Starting service with 'gunicorn app:app'
Aug 8 09:01:08 PM  [2023-08-09 01:01:08 +0000] [51] [INFO] Starting gunicorn 21.2.0
Aug 8 09:01:08 PM  [2023-08-09 01:01:08 +0000] [51] [INFO] Listening at: http://0.0.0.0:10000 (51)
Aug 8 09:01:08 PM  [2023-08-09 01:01:08 +0000] [51] [INFO] Using worker: sync
Aug 8 09:01:08 PM  [2023-08-09 01:01:08 +0000] [54] [INFO] Booting worker with pid: 54
Aug 8 09:01:09 PM  Your service is live 🎉
Aug 8 09:01:10 PM  127.0.0.1 - - [09/Aug/2023:01:01:10 +0000] "GET / HTTP/1.1" 200 225 "-" "Go-http-client/2.0"
```

Now, let's click our Render-provided URL at the top left again. We should now see the return JSON showing the activities currently in our database and the number count of activities.

```
{"activity_count":"0","activities":[]}
```

Note: If you are using an existing PostgreSQL database, any previously added activities to this database table will display.

Next, try inserting a new activity by accessing the following endpoint in the browser window: `https://<UNIQUE-RENDER-APP-NAME.onrender.com/insert_activity`. There should be new JSON output that shows the name of the activity that was successfully inserted into the database.

```
{"status":"success","message":"Activity \"Pot some plants and put them around your house\"
inserted successfully"}
```

Congratulations! We've just successfully deployed our back-end application using a Render Web Service and PostgreSQL database!

Wrap Up

As we've seen, Render provides a straightforward tool for creating and configuring our own PostgreSQL database that we can then use within our deployed source code via the setting of environment variables. To recap, in this article, we covered how to do the following:

- Fork a sample back-end code repository in GitHub
- Connect a code repository in Render to create a web service
- Create and configure a PostgreSQL database instance
- Create and add environment variables to a deployed web service

Now that we have completed the deployment of a back-end application, try deploying another application!