Deploying a Full-Stack Application with Render

Learn how to deploy a full-stack application using Render!

Introduction

The development of a full-stack application involves creating the front-end components, back-end components, communication between the two, and a resource that enables data storage, retrieval, and processing.

In this tutorial, we will practice using Render to deploy a full-stack application that uses HTML, CSS, JavaScript, React, PostgreSQL, and Node.js. To deploy our application, we will be:

- **Forking the Full-Stack Application Code Repository**: fork a sample full-stack application to use for our deployment
- **Configuring and Deploying with Render**: configure a Render web service 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!

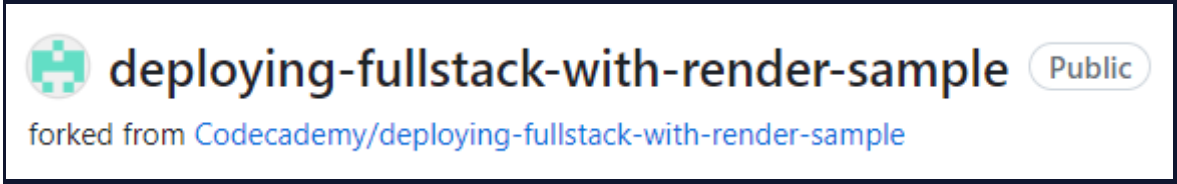
Forking the Full-Stack Application Code Repository

To get started with deploying a full-stack application with Render, we will use an existing full-stack application that provides an interactive interface for selecting what randomized activities we want to do today. This application uses a Node.js-based front-end for serving the user interface and Node on the back-end for handling the API logic between the two components. The back-end connects with an external API to retrieve the randomized activity options

that get displayed to the user. A PostgreSQL database is used to keep track of the selected activities.

For this tutorial, we will be using the `deploying-fullstack-with-render-sample` full-stack application. We'll need to connect our GitHub account to Render and then connect our desired repository. But, it is important to note that we only connect one GitHub account to one Render account. 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`.



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

Notice that the repository has a file called `app.js`, which holds the main front-end code used to create the user interface. Next, there is a `client/src/services` directory that contains a file, `activities.js`, which stores the source code needed to associate the front-end activity with the back-end API call. Finally, we see another important file called `requests.js` within the `services` directory which is used to retrieve data from and update the PostgreSQL database depending on the front-end action.

We'll see two additional files called `package-lock.json` and `package.json` in the repository that will be used to specify which packages and versions are needed for our application to build and deploy successfully.

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 "New +" button to create and configure our deployment. From the dropdown, select the "Web Service" option to

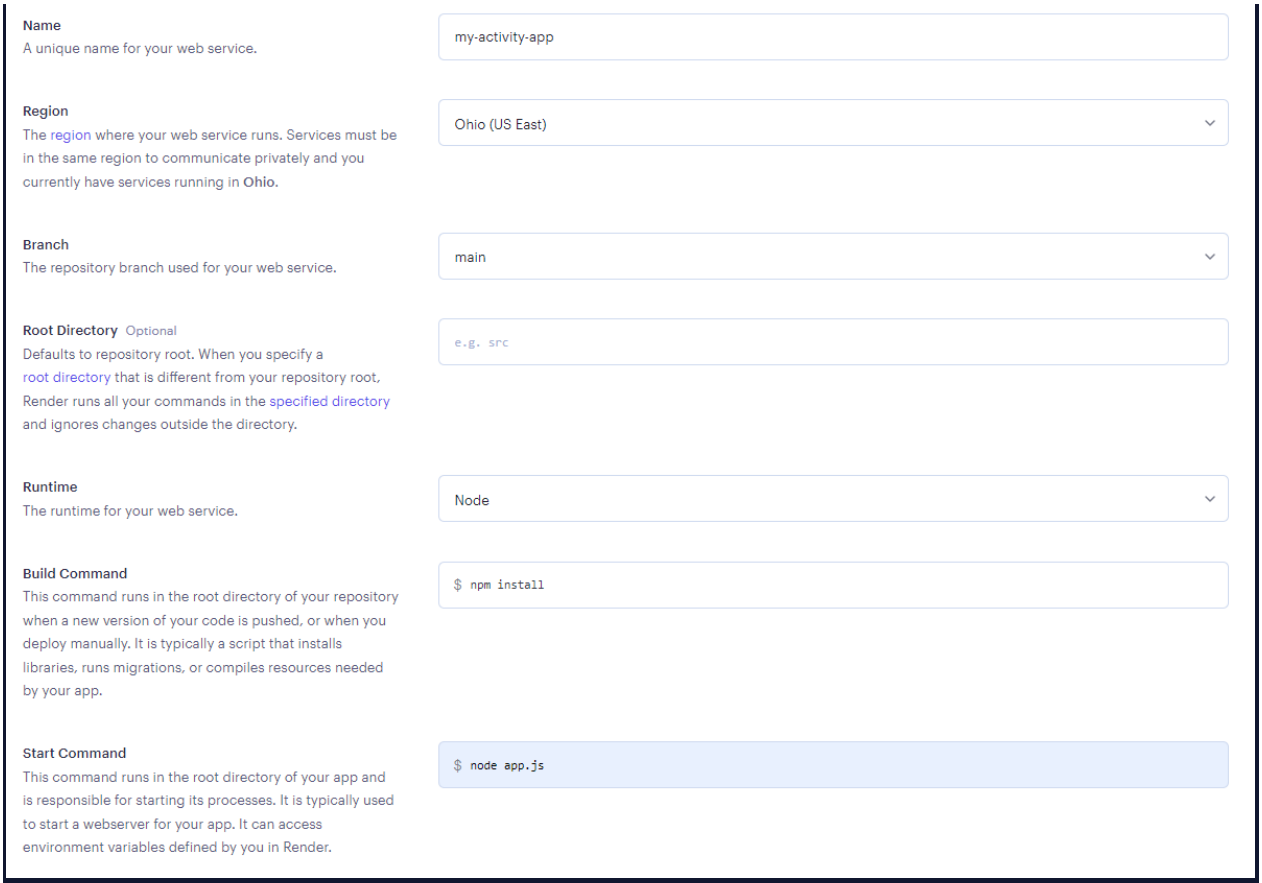
deploy our application to a web server. For a recap on how to deploy a web service in Render, check out Deploying a Simple App with Render.

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.

Configure the **Name**, **Region**, and **Branch** settings for the deployment based on personal preferences and forked repository setup. Next, leave the **Root Directory** blank so that we use the repository's default root directory. As mentioned, the **Runtime** has been pre-set to `Node`. For our **Build Command**, looking over the repository's code (specifically `package.json` we are using `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, use `node app.js` to start the Node server and serve our user interface using our `app.js` front-end file.



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

Note: 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

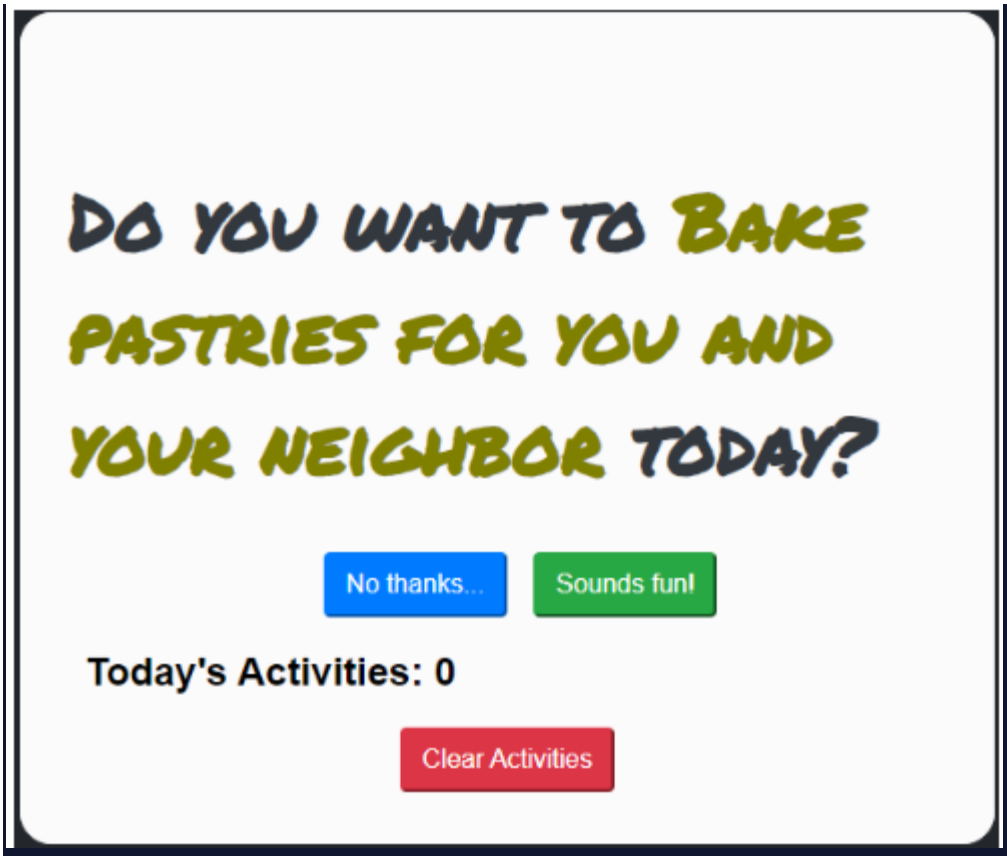
At the bottom of the web service configuration page, click on the "Create Web Service" button to start building and deploying the application.

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 commands being run and eventually a notice that our service is running on our specified port!

```
Aug 8 07:29:59 PM  ==> Checking out commit bf99aeac564dfd381fa83ece2c60895b2424d3c4 in branch main
Aug 8 07:30:03 PM  ==> Using Node version 14.17.0 (default)
Aug 8 07:30:03 PM  ==> Docs on specifying a Node version: https://render.com/docs/node-version
Aug 8 07:30:03 PM  ==> Running build command 'npm install'...
Aug 8 07:30:06 PM  added 72 packages from 48 contributors and audited 72 packages in 1.891s
Aug 8 07:30:06 PM
Aug 8 07:30:06 PM  1 package is looking for funding
Aug 8 07:30:06 PM    run `npm fund` for details
Aug 8 07:30:06 PM
Aug 8 07:30:06 PM  found 3 high severity vulnerabilities
Aug 8 07:30:06 PM    run `npm audit fix` to fix them, or `npm audit` for details
Aug 8 07:30:07 PM  ==> Uploading build...
Aug 8 07:30:14 PM  ==> Build uploaded in 7s
Aug 8 07:30:14 PM  ==> Build successful 🎉
Aug 8 07:30:14 PM  ==> Deploying...
Aug 8 07:30:38 PM  ==> Using Node version 14.17.0 (default)
Aug 8 07:30:38 PM  ==> Docs on specifying a Node version: https://render.com/docs/node-version
Aug 8 07:30:38 PM  ==> Starting service with 'node app.js'
Aug 8 07:30:39 PM  App is running on 10000
```

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.

We'll also get confirmation from the green "Live" state above the console window. Then we can try accessing the application user interface using the Render-provided URL at the top left. Clicking the provided link should open a new browser tab where our full-stack application is running! Try experimenting with the user interface by clicking the **Sounds fun!** button to add a few activities. We should see our **Today's Activities** count increase as we click this button. But wait, the count doesn't seem to be increasing!



Recall in the Monitoring and Maintaining a Deployed Render App article that we can debug our deployed applications by viewing the **Logs** menu option on the web service dashboard.

Let's view the logs now to see if we can debug why our count is not increasing. In the log, there should be some errors that look similar to the following:

```
Jun 19 10:54:31 PM  Error: connect ECONNREFUSED 127.0.0.1:5432
Jun 19 10:54:31 PM      at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1146:16) {
Jun 19 10:54:31 PM    errno: -111,
Jun 19 10:54:31 PM    code: 'ECONNREFUSED',
Jun 19 10:54:31 PM    syscall: 'connect',
Jun 19 10:54:31 PM    address: '127.0.0.1',
Jun 19 10:54:31 PM    port: 5432
Jun 19 10:54:31 PM  }
```

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 `services/requests.js` file, we can see that we are attempting 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 on.

Looking at our `services/requests.js` file, 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`, that 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 for use in our code and can affect the behavior of our applications. Recall, in our `services/requests.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.

Free response

Explain some benefits of setting environment variables in Render when deploying an application.

Your response

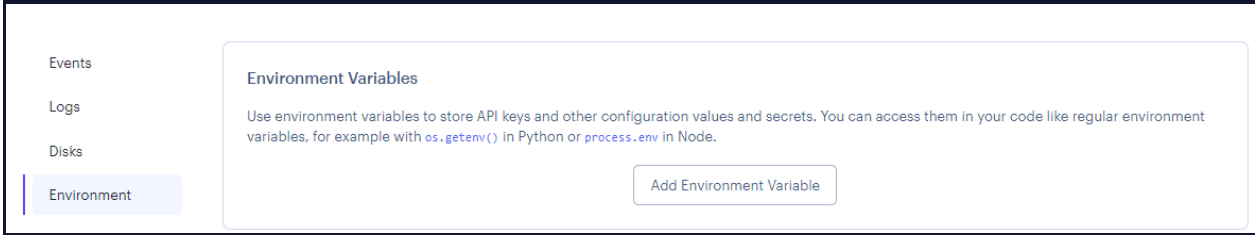
We can set up dedicated resources such as a database for activities like testing, without affecting the production database.

Our answer

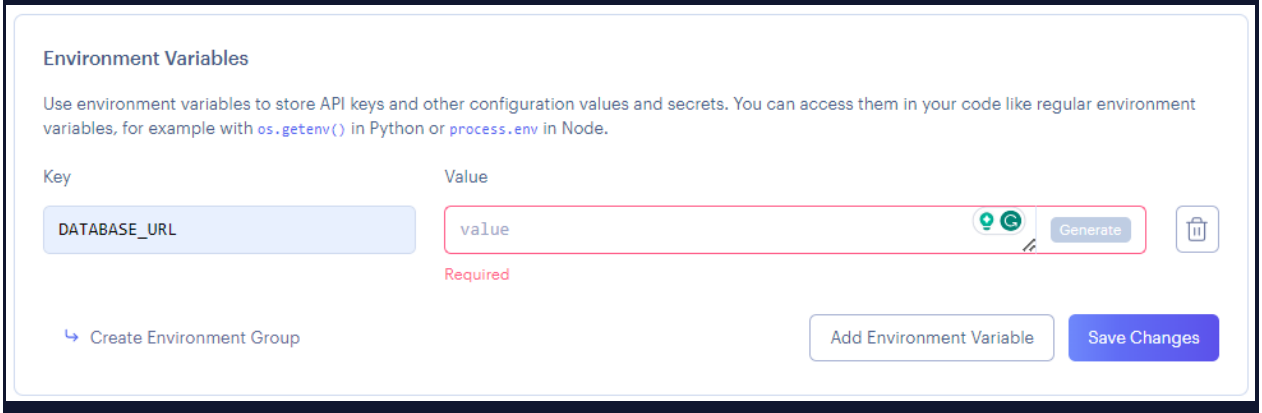
By setting and using environment variables, developers can streamline the deployment processes by configuring different environment variable values for different deployed environments.

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-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.



For the "Key" field, supply the `DATABASE_URL` name that our `services/requests.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.



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.

When we initially deployed the web service, we can find the port number in the **Logs** tab and see that it is running on port 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.

Verifying Deployment with Added Environment Variables

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

Now, let's click our Render-provided URL at the top left again and try hitting the "Sounds fun!" button. We should now see the "Today's Activities" count increase!

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

Wrap Up

As we've seen, Render provides an easy 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 full-stack 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

If you'd like to try to deploy another full-stack application, Render has additional full-stack applications that can be forked and deployed!