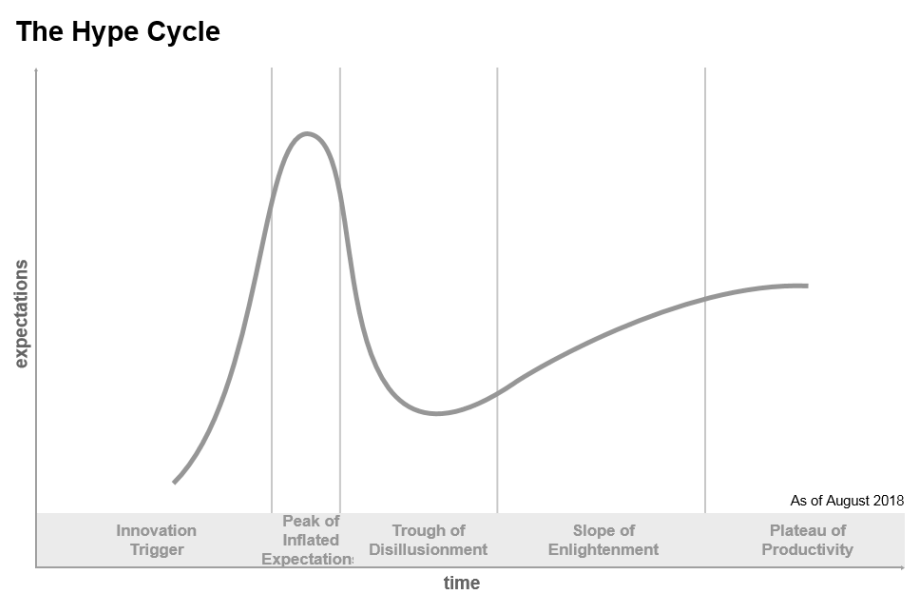
Dev Spotlight

Content Type	Ghostwritten Blog
Title	Time For Me To Fly… To Render (Part 1 of 2)
Client	Render - Ed Ropple
Length	~1800 words
Target Reader	Developer/Engineer
Sources	Outline
Status	Customer Draft
Notes	

Time For Me To Fly… To Render

[Heroku has eliminated their free plans, so I'm migrating to Render for my prototype products and services. Let's see how easy it is to convert to Render PaaS.]

The Gartner hype cycle, illustrated below, can be applied to most aspects of technology:



As new innovations enter their respective cycles, expectations are eventually realized—leading to some level of adoption. The goal for every innovation is to reach the plateau of productivity where consumers have determined that the reward of adopting the innovation far outweighs any known risks.

At the same time, there is a point where the plateau of productivity begins to diminish, leading to an exodus away from that innovation. One simple example would be pagers (or beepers), which were common before mobile phones/devices reached the plateau of productivity.

As technologists, we strive to deliver features, frameworks, products, or services that increase the plateau of productivity. The same holds true for the ones that we use.

Recently, I felt like my current hosting platform began falling off the plateau of productivity. In fact, a recent announcement made me wonder if it was time to consider other options.

Since I had a positive experience using the Render PaaS, I wanted to look at how easily I could convert one of my Heroku applications, adopt PostgreSQL, and migrate to Render. I'm describing that journey in this two-part series:

- Part 1: We'll focus on migrating my backend services (Spring Boot and ClearDB MySQL).
- Part 2: We'll focus on porting and migrating my frontend Angular client.

Why I Chose Render

If you have never heard of Render before, check out some of my previous publications:

- Using Render and Go for the First Time
- Under the Hood: Render Unified Cloud
- Purpose-Driven Microservice Design
- Launch Your Startup Idea in a Day
- How I Used Render To Scale My Microservices App With Ease

What I find exciting about Render is that they continue to climb the slope of enlightenment while actively providing a solid solution for adopters recognizing the plateau of productivity. As I've noted in my articles, Render offers a "Zero DevOps" promise. This perfectly aligns with my needs since I don't have the time to focus on DevOps tasks.

The Heroku platform has several things that I am not too fond of:

- Daily restarts led to unexpected downtime for one of my services.
- Entry level (all I really need) Postgres on Heroku allows for four hours of downtime per month.
- Pricing levels, from a consumer perspective, don't scale well.

From a pricing perspective, I am expecting to see significant cost savings after migrating all of my applications and services from Heroku to Render. What is more amazing is that I am getting **better memory and CPU for that price,** with linear scaling as my application footprint needs to grow.

Converting a Single Service

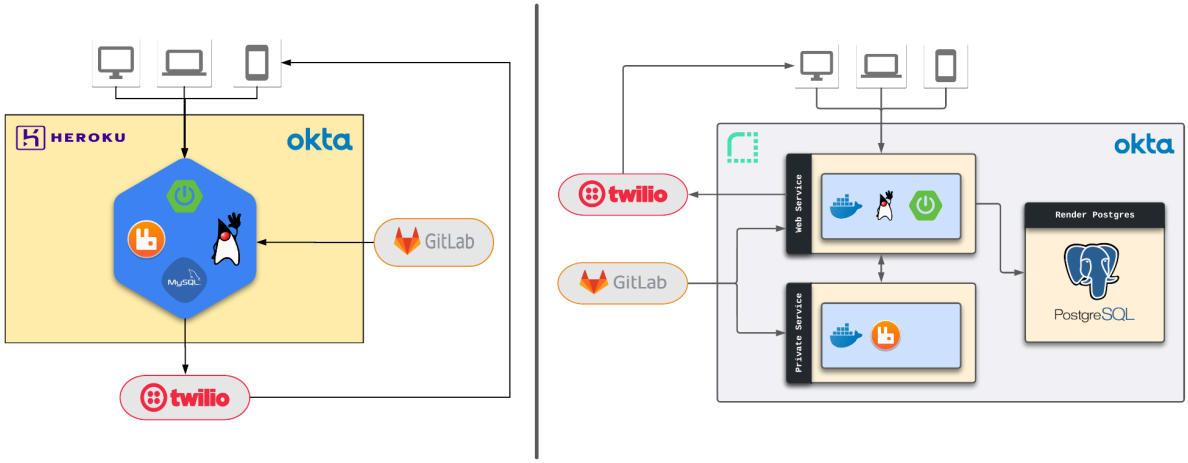
As I noted above, this is part one of a two-part series, and I'll focus on the service tier in this article. The service I want to convert has the following attributes:

- Spring Boot RESTful API Service
- Heroku CloudAMQP (RabbitMQ) Message Broker
- Heroku ClearDB (MySQL) Database (single schema)
- Okta Integration

On the Render PaaS side, the new service design will look like this:

- Render Web Service hosting Spring Boot RESTful API Service (via Docker)
- Render Private Service hosting RabbitMQ Message Broker (via Docker)
- Render Postgres with the ability for multiple schemas to exist
- Okta Integration

Below is a side-by-side comparison of the two ecosystems:



My high-level plan of attack for the conversion is as follows:

Prepare Heroku for Conversion

Before getting started, it is recommended to put all existing Heroku services into Maintenance Mode. This will prohibit any consumers from accessing the applications or services. While the source code should already be backed up and stored in a git-based repository, it is a good idea to make sure a database backup has been successfully created.

Conversion of Heroku Services

From a conversion perspective, I had two items to convert: the service itself and the ClearDB (MySQL) database.

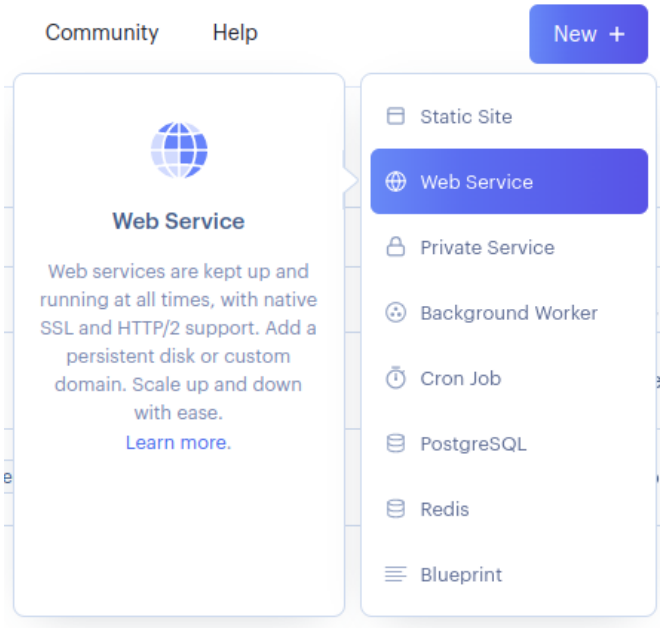
The conversion of my Spring Boot RESTful service did not involve much work. In fact, I was able to leverage the approach I used for a previous project of mine.

For the database, I needed to convert from MySQL to PostgreSQL. My goal was to use Render's Heroku Migrator to easily migrate Heroku Postgres to Render Postgres, but I needed to convert from MySQL to PostgreSQL first.

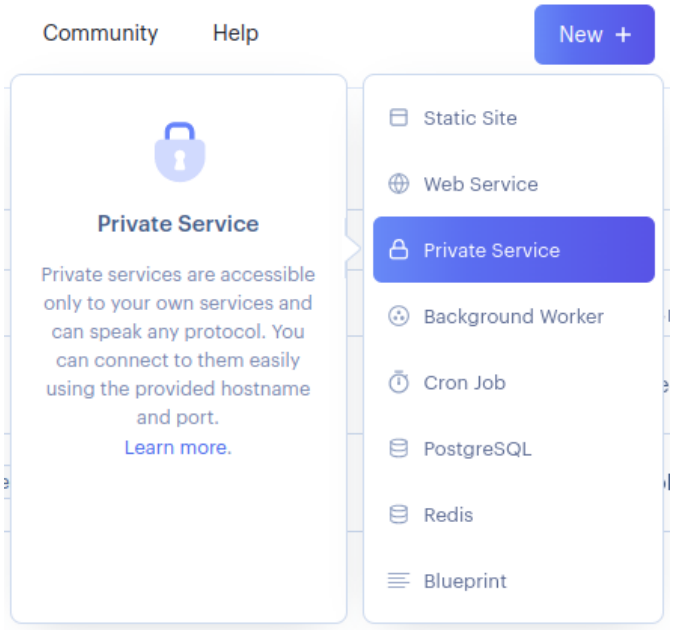
Initially, I started down the pgloader path, which seemed to be a common approach for the database conversion. However, using my M1-chip MacBook Pro led to some unexpected issues. Instead, I opted to use NMIG to convert MySQL to PostgreSQL. For more information, please check out the "Highlights From the Database Conversion" section below.

Create Services in Render

After converting the database and the Spring Boot RESTful service running inside Docker, the next step was to create a Render Web Service for the Spring Boot RESTful API service. This was as easy as creating the service, giving it a name, and pointing to the appropriate repository for my code in GitLab.



Since I also needed a RabbitMQ service, I followed these instructions to create a RabbitMQ Private Service running on Render. This included establishing a small amount of disk storage to persist messages that have not been processed.

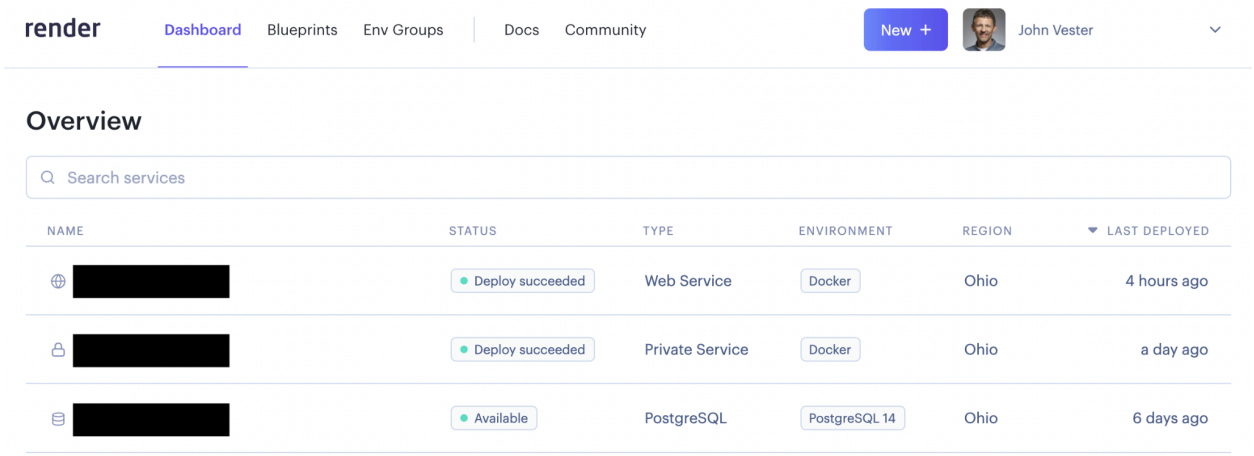


Finally, I created the necessary environment variables in the Render Dashboard for both the Spring Boot RESTful API service and the RabbitMQ message broker.

Initialize and Validate the Services

The next step was to start my services. Once they were running and the APIs were validated using my Postman collection, I updated my client application to point to the new Render service location.

Once everything was up and running, my Render Dashboard appeared as shown below:

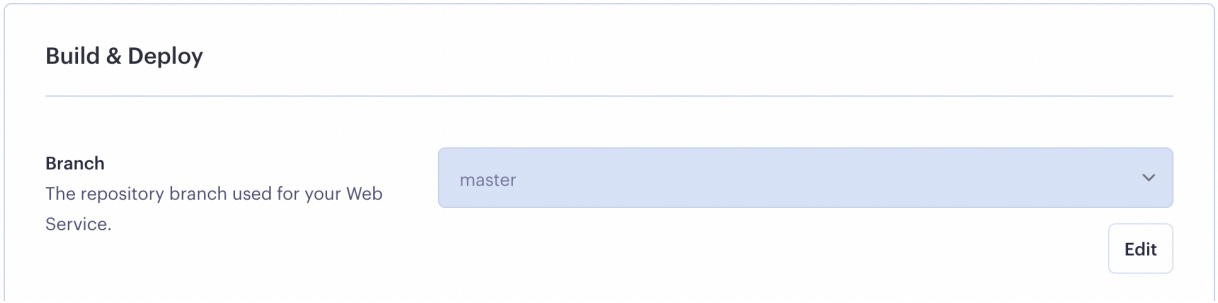


Next Steps

All that remained at this point was to delete the databases still running on Heroku and remove the migrated services from the Heroku ecosystem.

When using Heroku, any time I merged code into the `master` branch of my service repository, the code was automatically deployed, provided I used GitLab CI/CD to deploy to Heroku in my source repository.

However, there is no need to add code to the source file repository with Render. I simply needed to specify the **Build & Deploy Branch** in the Render Dashboard for the service:



I love the Zero DevOps promise.

Highlights From the Database Conversion

By following the steps above, the conversion from Heroku to Render was smooth and successful. The biggest challenge for me was the conversion of data. At a high level, this mostly boiled down to a series of commands executed from the terminal of my MacBook Pro.

First, I started a local Postgres instance via Docker:

```
docker run --publish 127.0.0.1:5432:5432 --name postgres -e POSTGRES_PASSWORD=dbo
-d postgres
```

Next, I created a database called “example” using the following command (or pgAdmin):

```
createdb example
```

For converting my ClearDB (MYSQL) instance running on Heroku to my example Postgres database running locally, I used NMIG, which is a Node.js-based database conversion utility.

After installing NMIG, I set up the `config.json` file with database endpoint information and credentials, and then I ran:

```
/path/to/nmig$ npm start
```

Next, I backed up the data to a file using the following command:

```
pg_dump -Fc --no-acl --no-owner -h localhost -U postgres example > example.dump
```

Rather than go through the hassle of creating a signed URL in AWS, I just used the pgAdmin client to import the backup into a newly created Postgres instance on Heroku.

With the Postgres instance running and data validated, I created a new Postgres database on the Render PaaS. Then all I had to do was issue the following command:

```
pg_restore --verbose  --no-acl --no-owner -d
postgres://username:password@hostname.zone-postgres.render.com/example example.dump
```

Lessons Learned Along the Way

Looking back on my conversion from Heroku to Render, here are some lessons I learned along the way:

- I had a minor issue with the Postgres database updating the date/time value to include the DST offset. This may have been an issue with my original database design, but I wanted to pass this along. In my case, the impacted column is only used for Date values, which did not change for me.
- I included a database column named `END` in one of my tables, which caused a problem when either Postgres or Hibernate attempted to return a native query. The service saw the `END` column name and injected it as a SQL keyword. I simply renamed the column to fix this issue, which I should have known not to do in the first place.
- With Render, I needed to make the RabbitMQ service a Private Service because the Web Service option does not expose the expected port. However, with this approach, I lost the ability to access the RabbitMQ admin interface, since Private Services are not exposed externally. It looks like Render plans to address this feature request.

All in all, these minor hurdles weren't significant enough to impact my decision to migrate to Render.

Conclusion

The most important aspect of Gartner's plateau of productivity is providing products, frameworks, or services that allow consumers to thrive and meet their goals. The plateau of productivity is not intended to be flashy or fashionable—in a metaphorical sense.

When I shared this conclusion with Ed, a Developer Advocate at Render, his response was something I wanted to share:

> *"Render is pretty avowedly not trying to be 'fashionable.' We're trying to be unsurprising and reliable."*

Ed's response resonated deeply with me and reminded me of a time when my former colleague told me my code came across as "boring" to him. His comment turned out to be the greatest compliment I could have received. You can read more here.

In any aspect of technology, the decision on which provider to select should always match your technology position. If you are unsure, the Gartner hype cycle is a great reference point, and you can get started with a subscription to their service here.

I have been focused on the following mission statement, which I feel can apply to any IT professional:

> *"Focus your time on delivering features/functionality that extends the value of your intellectual property. Leverage frameworks, products, and services for everything else."*
>
> *- J. Vester*

When I look at the Render PaaS ecosystem, I see a solution that adheres to my mission statement while residing within my hype cycle preference. What makes things better is that I fully expect to see a 44% savings in my personal out-of-pocket costs—even more as my services need to scale vertically.

For those considering hosting solutions, I recommend adding Render to the list of providers for review and analysis. You can get started for free by following this link.

The second part of this series will be exciting. I will demonstrate how to navigate away from paying for my static client written in Angular and take advantage of Render's free Static Sites service using either Vue or Svelte. Which framework will I choose … and why?

Have a really great day!