Introduction

The project files are located in the `8-serverless` folder.

Not so long ago I launched Laracheck which is a code review tool. This is the way it works:

- You install it on GitHub as an app
- You open a PR
- It reviews your code and leaves comments on the PR

So it's not a package that you can install in your repository but a stand-alone application with APIs and a database.

There are ~15 checks it can run. To name a few:

- N+1 query detection
- Incorrect code dependency detection
- Missing DB indices or foreign keys detections
- etc

The whole application is "serverless." It runs on a PaaS (Platform-as-a-service) solution and uses serverless functions and a managed database. In this essay, I'm going to show you how it's designed, why I choose these solutions/technologies, and what are the advantages and disadvantages.

In the first part, I'm going to talk about PaaS and functions in general. Then I'll describe the exact design of Laracheck.

Platform-as-a-service

As you might guess, I'm using DigitalOcean to deploy this project. App Platform is their PaaS service. Other providers also have similar solutions:

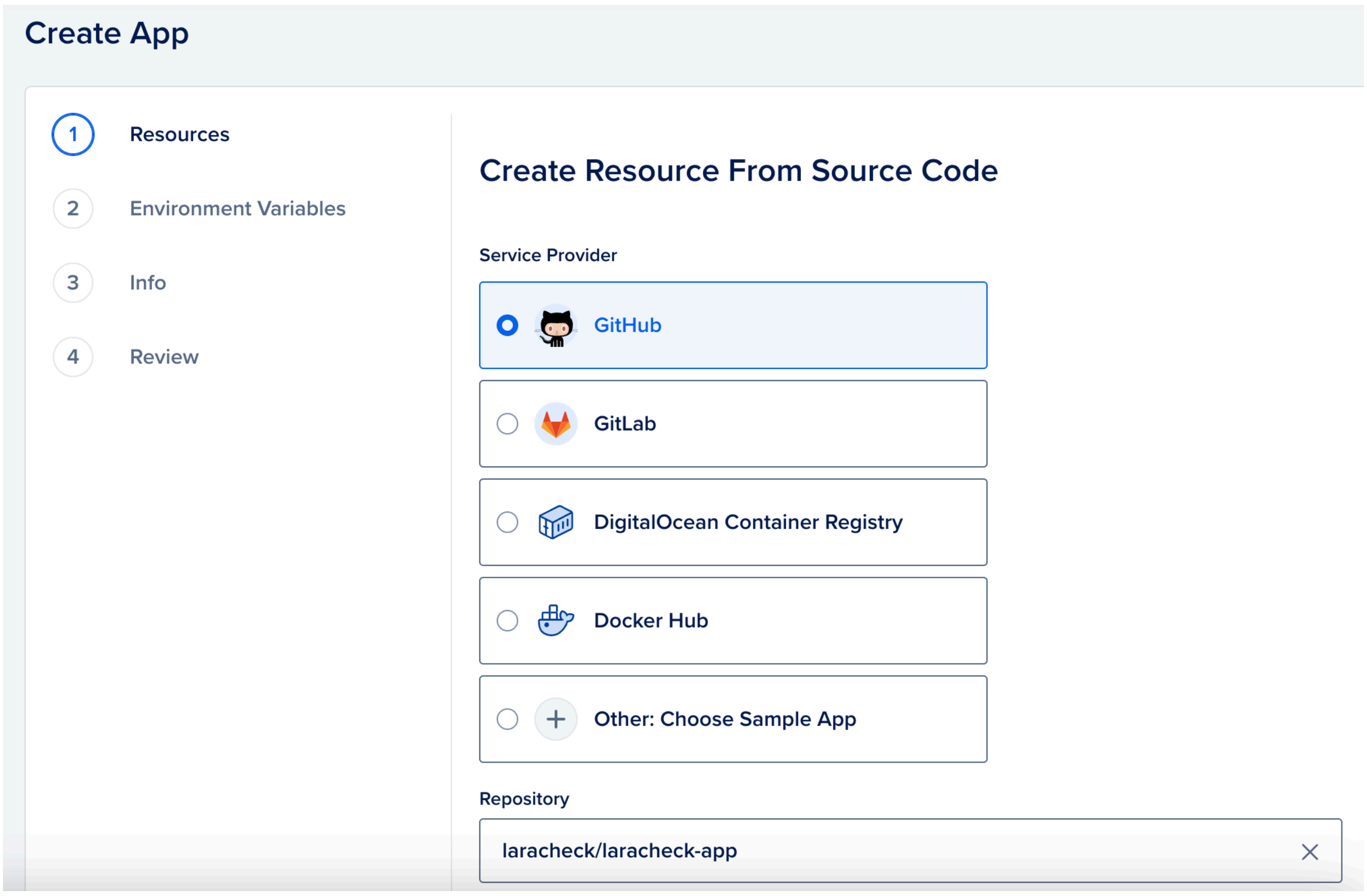
- Amazon AWS App Runner
- Google GCP App Engine
- Microsoft Azure App Service

The main feature of these services is that you don't need to create servers at all. You don't even need to write Dockerfiles. You just give them a GitHub repository, set up your environment variables, the scaling options and capacity you need, and you're done with your infrastructure. It builds Docker images from your code and then runs them on nodes managed by your providers. You can scale up and down the components (such as API or frontend) and choose the node size.

In DigitalOcean these are the important terms:

- App refers to your whole application
- Component refers to one component such as API or worker

When you create a new app you need to choose the source of your first component:


Create App

1 Resources
2 Environment Variables
3 Info
4 Review

Create Resource From Source Code

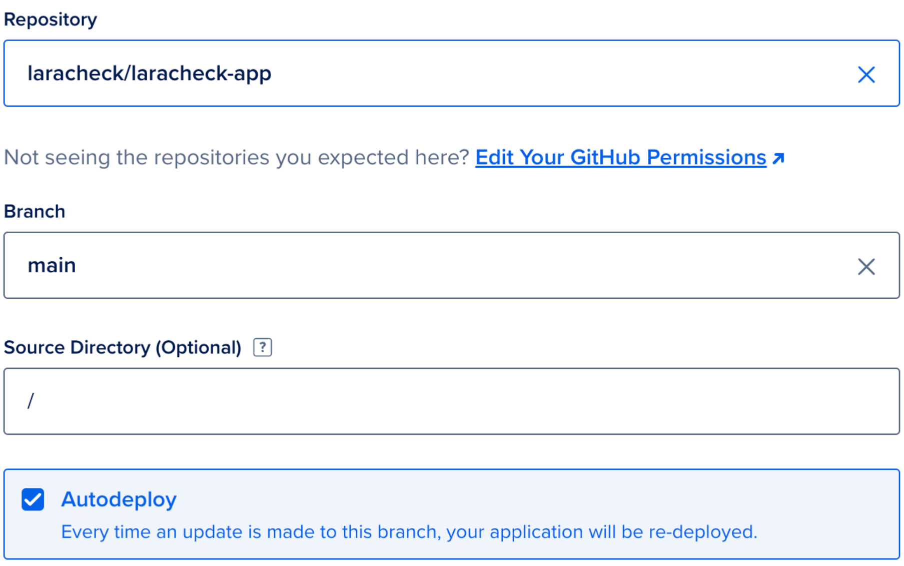
Service Provider

- [✓ GitHub]
- GitLab
- DigitalOcean Container Registry
- Docker Hub
- Other: Choose Sample App

Repository

laracheck/laracheck-app

Then you can select the source branch:


Repository

laracheck/laracheck-app

Not seeing the repositories you expected here? Edit Your GitHub Permissions

Branch

main

Source Directory (Optional)

/

☑ Autodeploy
Every time an update is made to this branch, your application will be re-deployed.

Using the `Source Directory` option you can deploy from monorepos as well.

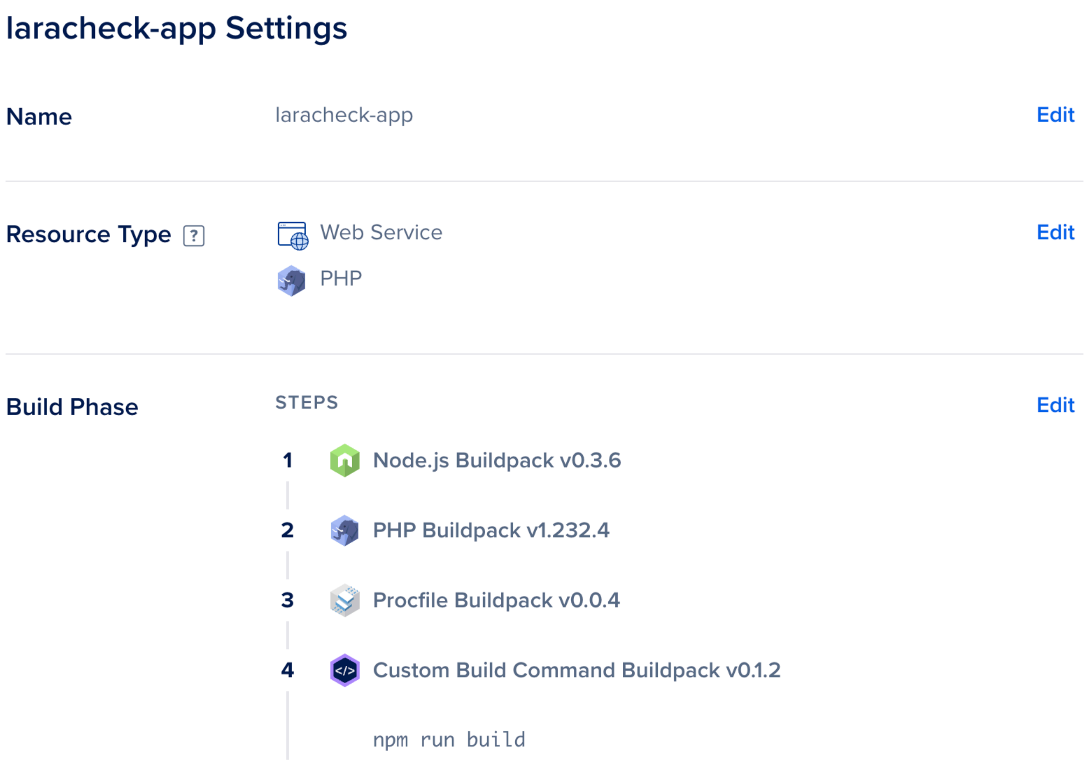
In this case, I have one repository which is an MVC app. Meaning, there's no separate API and SPA frontend but only a Laravel app with Blade views. So far in the book, we only talked about APIs so now we can deploy a more traditional SSR application as well.

Here's what I used to do:

- I create two separate applications for the project
- One of them is the staging environment and the source branch is `develop`
- The other one is for production and the source is `main`

There's an option called `Autodeploy`. If you enable it the application gets updated every time when you push new commits to the source repository. It's a great option because it makes development and deployment pretty fast. However, this way you cannot have a pipeline. I mean, you could, but the app is updated no matter what. Whenever you push a commit, it's going to be shipped.

On the next page, you can see that DigitalOcean detected that it's a PHP application and selected the right build packs. This is how it's going to build a Docker image from the source.

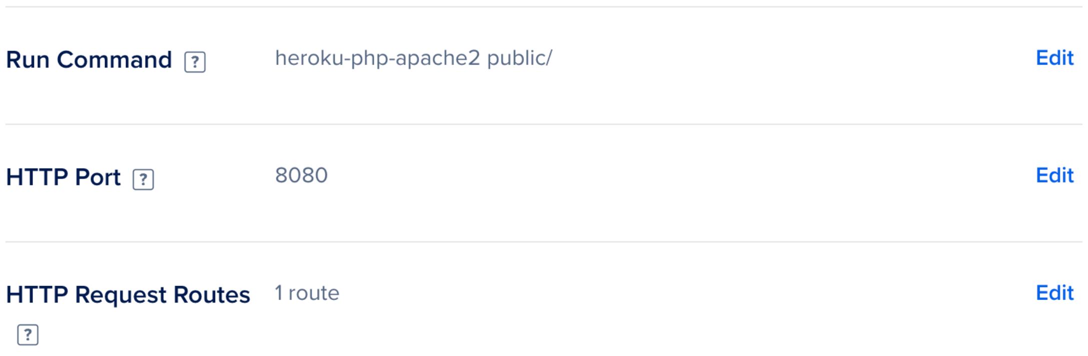


laracheck-app Settings

Name	laracheck-app	Edit
Resource Type ?	Web Service PHP	Edit
Build Phase	STEPS 1 Node.js Buildpack v0.3.6 2 PHP Buildpack v1.232.4 3 Procfile Buildpack v0.0.4 4 Custom Build Command Buildpack v0.1.2 `npm run build`	Edit

As you can see, App Platform detected that we have an MVC application so it added `npm run build` to the build phase. When it builds a Docker image from the source it'll run `npm run build` so all the assets are ready to use.

In the next section we can define the run command and some HTTP-related settings:



Run Command ?	heroku-php-apache2 public/	Edit
HTTP Port ?	8080	Edit
HTTP Request Routes ?	1 route	Edit

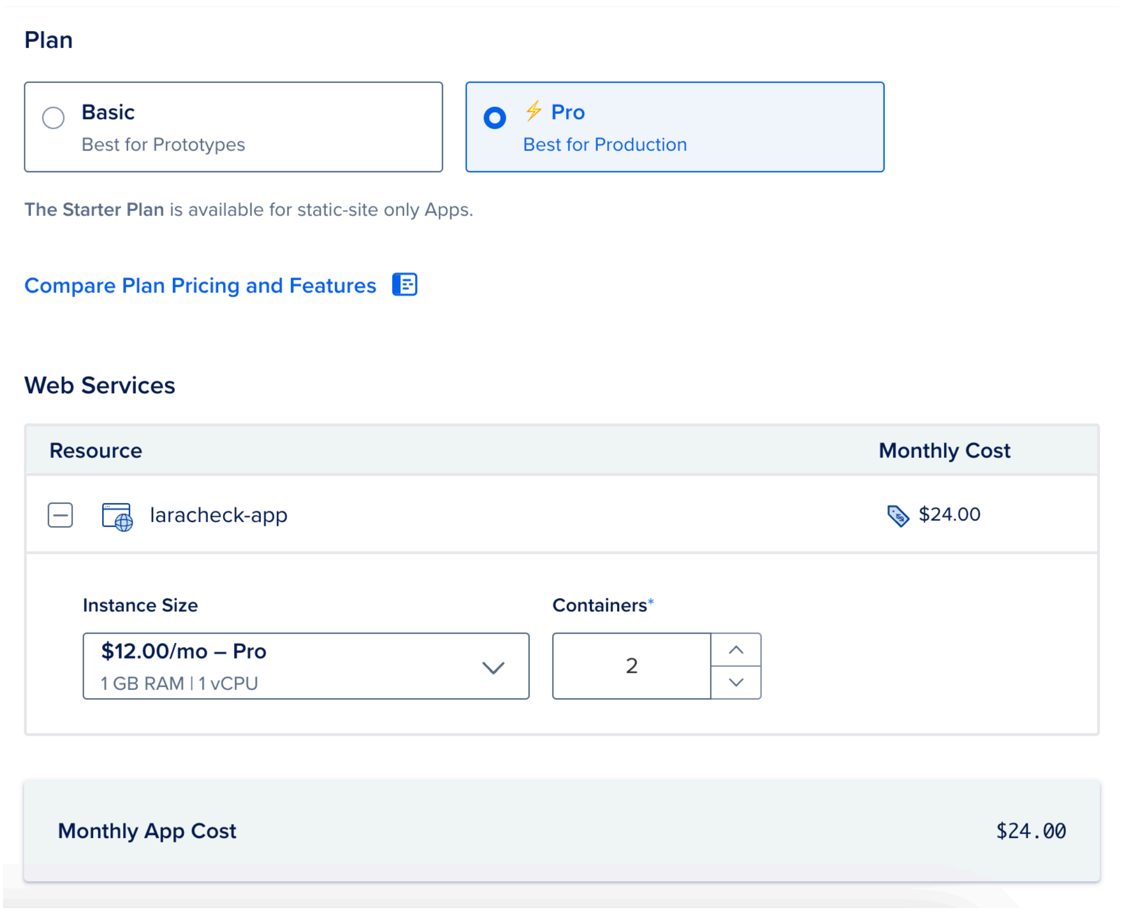
The basic run command serves the Laravel app via a web server. We can add a few extra commands to it:

```
php artisan optimize
heroku-php-apache2 public/
```

This is the same thing we did with Kubernetes but we ran this command in the Dockerfile.

HTTP port is an internal port for the other components. With HTTP request routes you can handle routing. So if you have an API you can write `/api`. This means that the component you're configuring will only accepts requests coming to `myapp.com/api`. In this case, I need this component to serve the frontend as well so I leave the default `/` route.

On the next page, you can set plan and scaling-related parameters:



Plan

( ) Basic
Best for Prototypes

[✓ Pro
Best for Production]

The Starter Plan is available for static-site only Apps.

Compare Plan Pricing and Features

Web Services

Resource	Monthly Cost
laracheck-app	$24.00

Instance Size: $12.00/mo – Pro, 1 GB RAM | 1 vCPU

Containers*: 2

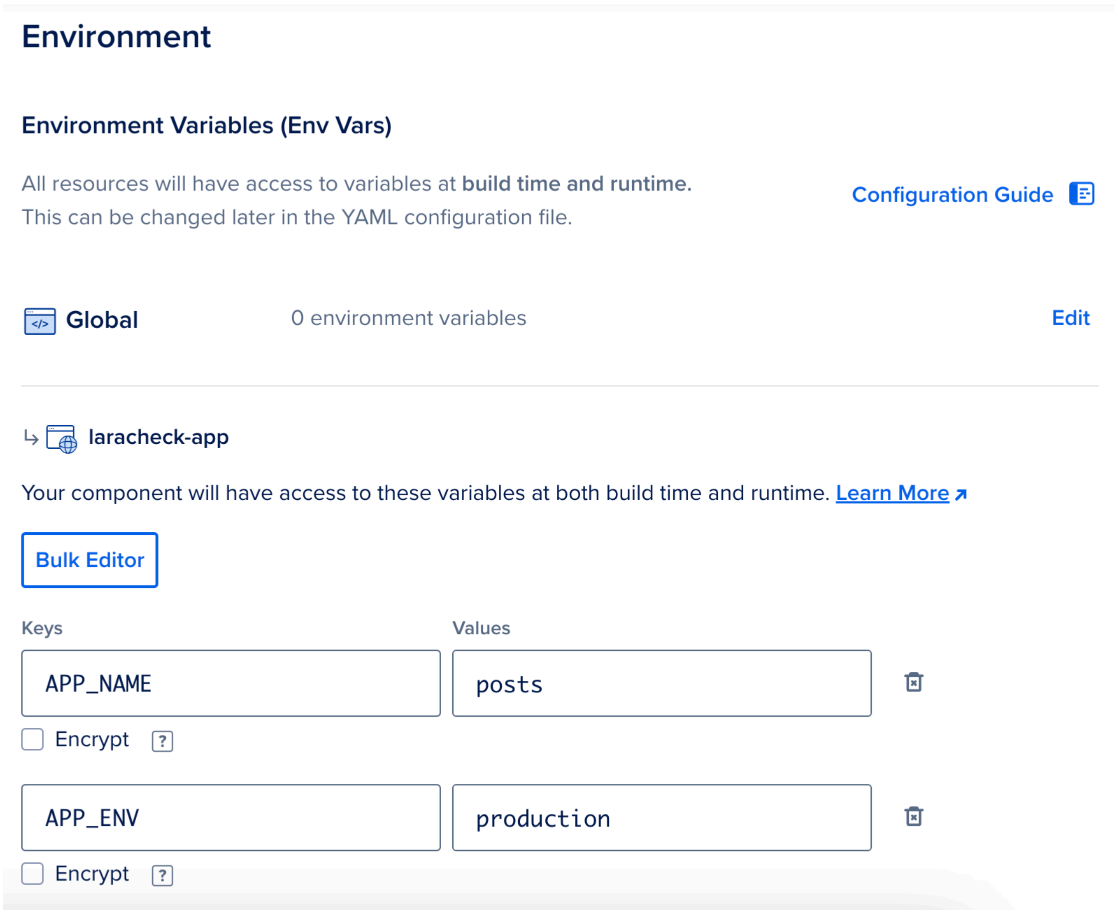
Monthly App Cost $24.00

Only pro plans can run multiple replicas from the same component.

On the next page, you can set environment variables. There are two kinds of variables:

- Global is available in the whole app for every component.
- Local variables can be only used by a specific component.

If you have project-related environment variables such as a `ROLLBAR_TOKEN` or `AWS_ACCESS_KEY` set them as globals so you don't need to do it again as you add more components to the app.


Environment

Environment Variables (Env Vars)

All resources will have access to variables at **build time and runtime.**
This can be changed later in the YAML configuration file.

Configuration Guide

Global 0 environment variables Edit

laracheck-app

Your component will have access to these variables at both build time and runtime. Learn More

Bulk Editor

Keys	Values
APP_NAME	posts
APP_ENV	production

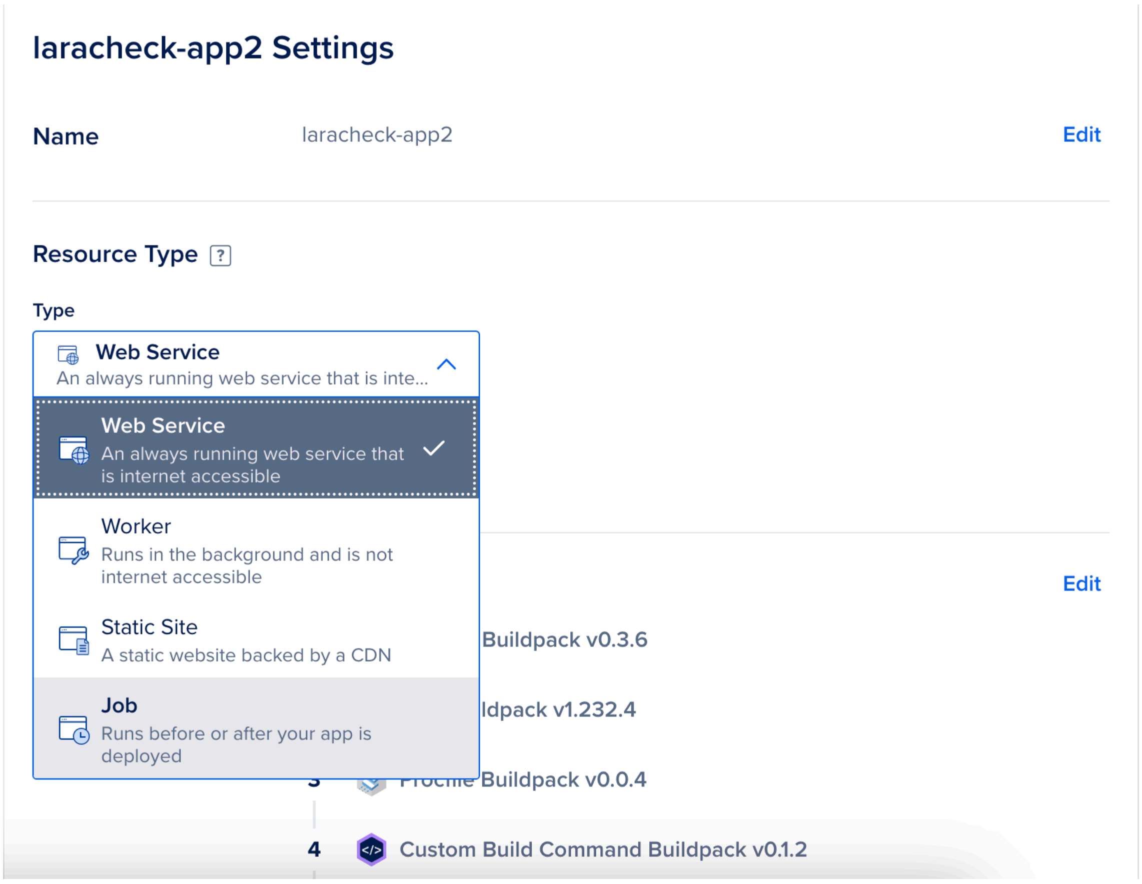
Encrypt

Encrypt

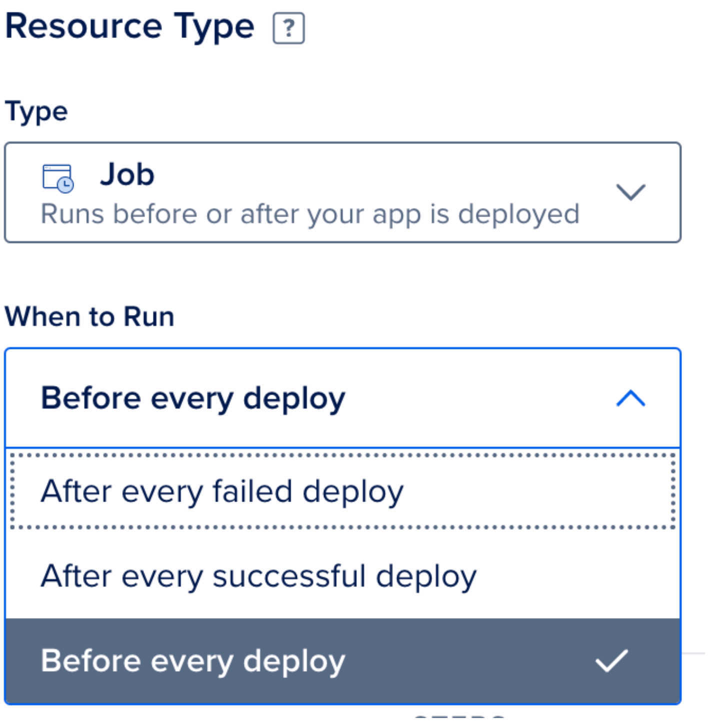
You can use an existing database server or you can create a new one exclusively for this application. I created a new database in my existing server. Either way, you'll get your connection details that can be set as env variables.

After that you're ready. The application is being built and then deployed. Your app can be accessed on an URL such as `https://your-app-pp5hs.ondigitalocean.app/`

Of course, it won't work yet. We need to migrate the database first. To do so we need similar solutions to Kubernetes or Swarm. We need a new component that runs after every deployment and executes `php artisan migrate`. Fortunately, DigitalOcean provides such components. You can add a new one from the same repository, but in the resource type options choose `Job`:


laracheck-app2 Settings
Name
laracheck-app2
Edit
Resource Type
Type
Web Service
An always running web service that is inte...
Web Service
An always running web service that is internet accessible
Worker
Runs in the background and is not internet accessible
Static Site
A static website backed by a CDN
Job
Runs before or after your app is deployed
Edit
Buildpack v0.3.6
ldpack v1.232.4
Buildpack v0.0.4
4
Custom Build Command Buildpack v0.1.2

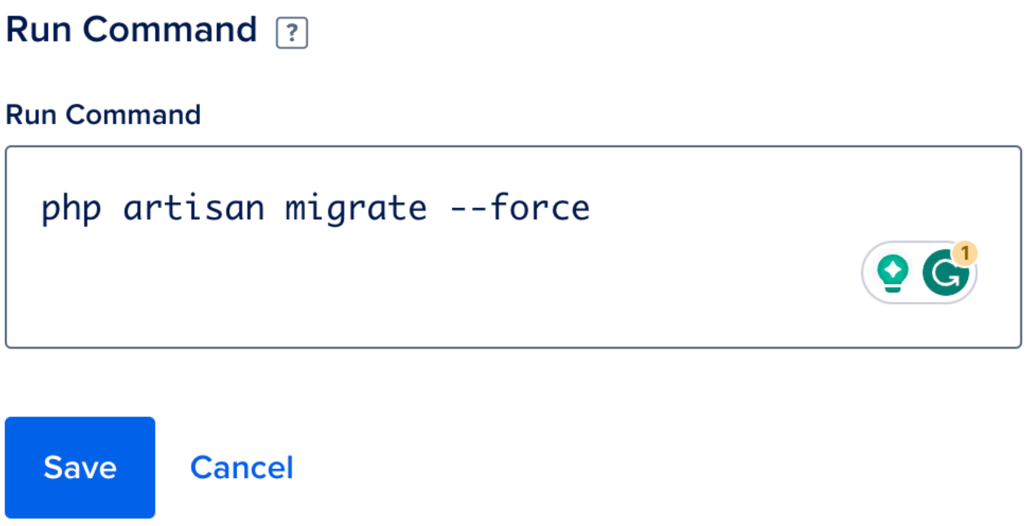
A job has another option called `When to run`:


Resource Type
Type
Job
Runs before or after your app is deployed
When to Run
Before every deploy
After every failed deploy
After every successful deploy
Before every deploy

Let's talk about the 3 options:

- After every failed deploy. It's out of the question right now but it can be useful if you want to send some notifications or do some other "clean-up" tasks after a deployment failed.
- Before every deploy. It sounds better. But if you choose this and your deployment fails your database will have the new schema but your code rolls back to the previous version. So your app is going to be in an inconsistent state. Which can cause bugs.
- After every successful deploy. That's the one we need. If the new version is rolled out successfully we can run the migrations as the last step. Just as running `php artisan migrate` was one of the last steps in the deploy script when we didn't even use Docker and containers.

And now the run command can be set to `php artisan migrate --force`:



```
php artisan migrate --force
```

So now we have a component that runs after every deploy and migrates the database. It runs only in one replica which is key when migrating the DB. You can run multiple replicas from jobs but in this case, it doesn't make sense.

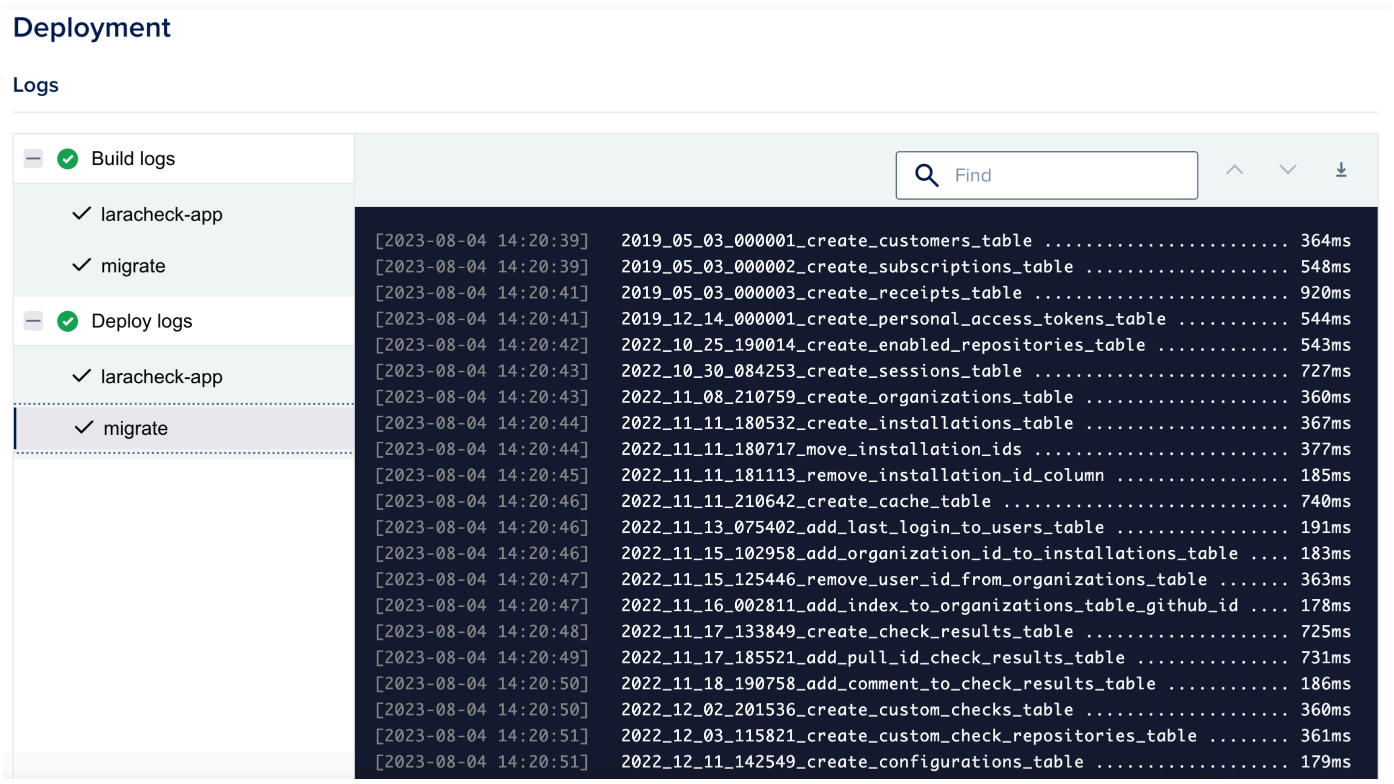
This is basically the same when we used a `Job` called `migrate` in Kubernetes:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: migrate
spec:
  backoffLimit: 2
  template:
    spec:
      restartPolicy: OnFailure
      containers:
        - name: migrate
          image: martinjoo/posts-api:$IMAGE_TAG
          command: ["sh", "-c", "php artisan migrate --force"]
          envFrom:
            - configMapRef:
                name: posts
            - secretRef:
                name: posts
```

And it's also very similar to the container called `update` in the Swarm stack:

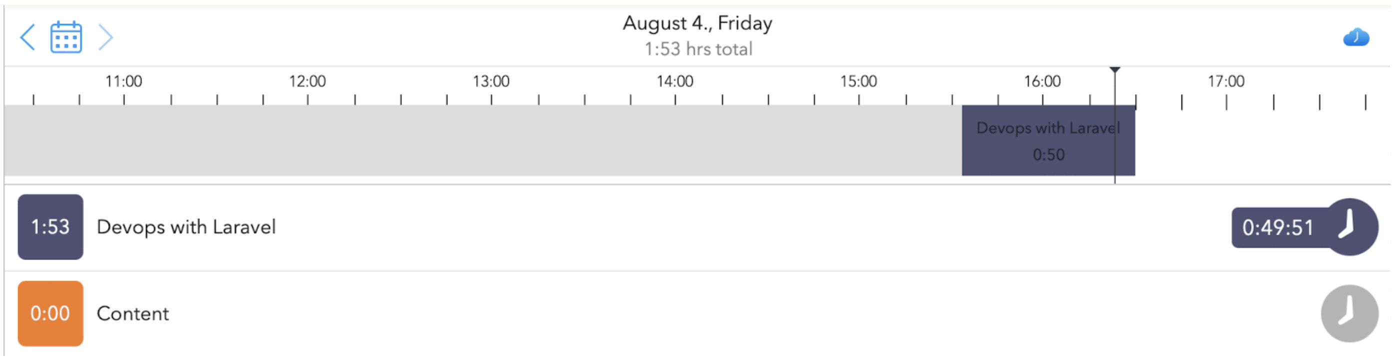
```yaml
update:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && /usr/src/update.sh"
  deploy:
    mode: replicated-job
```

If you now deploy the app it's going to be up and running. Under the `Activity` panel you can see the deployment logs:



```
[2023-08-04 14:20:39]  2019_05_03_000001_create_customers_table ...................... 364ms
[2023-08-04 14:20:39]  2019_05_03_000002_create_subscriptions_table .................. 548ms
[2023-08-04 14:20:41]  2019_05_03_000003_create_receipts_table ....................... 920ms
[2023-08-04 14:20:41]  2019_12_14_000001_create_personal_access_tokens_table .......... 544ms
[2023-08-04 14:20:42]  2022_10_25_190014_create_enabled_repositories_table ............ 543ms
[2023-08-04 14:20:43]  2022_10_30_084253_create_sessions_table ....................... 727ms
[2023-08-04 14:20:43]  2022_11_08_210759_create_organizations_table .................. 360ms
[2023-08-04 14:20:44]  2022_11_11_180532_create_installations_table .................. 367ms
[2023-08-04 14:20:44]  2022_11_11_180717_move_installation_ids ....................... 377ms
[2023-08-04 14:20:45]  2022_11_11_181113_remove_installation_id_column ............... 185ms
[2023-08-04 14:20:46]  2022_11_11_210642_create_cache_table .......................... 740ms
[2023-08-04 14:20:46]  2022_11_13_075402_add_last_login_to_users_table ............... 191ms
[2023-08-04 14:20:46]  2022_11_15_102958_add_organization_id_to_installations_table .... 183ms
[2023-08-04 14:20:47]  2022_11_15_125446_remove_user_id_from_organizations_table ....... 363ms
[2023-08-04 14:20:47]  2022_11_16_002811_add_index_to_organizations_table_github_id .... 178ms
[2023-08-04 14:20:48]  2022_11_17_133849_create_check_results_table .................. 725ms
[2023-08-04 14:20:49]  2022_11_17_185521_add_pull_id_check_results_table ............. 731ms
[2023-08-04 14:20:50]  2022_11_18_190758_add_comment_to_check_results_table .......... 186ms
[2023-08-04 14:20:50]  2022_12_02_201536_create_custom_checks_table .................. 360ms
[2023-08-04 14:20:51]  2022_12_03_115821_create_custom_check_repositories_table ........ 361ms
[2023-08-04 14:20:51]  2022_12_11_142549_create_configurations_table ................. 179ms
```

This is a screenshot from Hours which is a time-tracking app:



It took me **49 minutes and 51 seconds** to deploy a Laravel app with a database. And I also wrote down the whole process. And on top of that, the application runs in two replicas. I didn't create SSH keys, and didn't run any Linux-related commands. 0 lines of shell scripts were written. But I could already send you a link where my app is available. This is one of the **advantages** of PaaS services. Fast and easy deployments.

The **disadvantage** is that it costs you more money. You cannot just run your app, MySQL, and Redis on the same server anymore. You need a dedicated (probably managed) MySQL server, a Redis server, and then your app on App Platform. The migrate component also costs extra money. It's pretty cheap, but we have to pay for migrating our own databases now. Which is ridiculous if you think about it.

One thing to note: App Platform doesn't autoscale at the time. You need to manually add replicas by clicking the + button.

Serverless functions

The idea behind serverless functions is that you deploy atomic parts of your application as independent units. This unit is a function and it does only one thing and is usually exposed via HTTP. For example, you can image a single CRUD application with four functions:

- CreatePost
- UpdatePost
- DeletePost
- ListPosts
- ShowPost

Each connects to the same database and exposes the usual REST endpoints. But each of them contains only the necessary code.

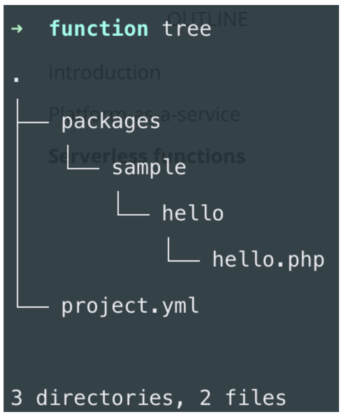
Let's see an example. First, you need to create a namespace on DigitalOcean which you can on the UI or in your terminal:

```
doctl serverless namespaces create --label example-namespace --region nyc1
```

Then you can init your project:

```
doctl serverless init --language php my-functions
```

This creates a new folder `my-functions` and initializes the project:



```
➜ function tree
.
├── packages
│   └── sample
│       └── hello
│           └── hello.php
└── project.yml

3 directories, 2 files
```

A package is a logical unit of organization. For example, a CRUD application might look like this:

```
- packages
  - posts
    - create
    - delete
  - comments
    - list
    - show
```

And inside the `hello.php` we have this:

```
function main(array $args): array
{
  $name = $args['name'] ?? 'stranger';

  return ['body' => "Hello $name!"];
}
```

Each of your functions has to follow some rules provided by DO:

- There has to be a `main` function which is the entry point
- It accepts an array that contains request parameters
- It returns an array that has a `body` key with your response

You can deploy the function with:

```
doctl serverless deploy my-function
```

Of course, you can version-control your functions, for now, it's just a quick introduction.

You can invoke it from the terminal:

```
doctl serverless functions invoke sample/hello
```

Which returns:

```
{
  "body": "Hello stranger!"
}
```

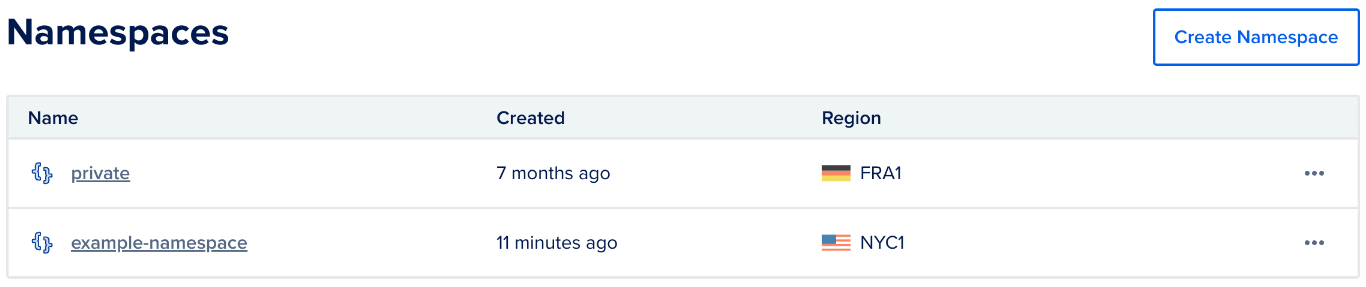
And you can also pass arguments from your terminal:

```
doctl serverless functions invoke sample/hello -p name:Joe
```

The result:

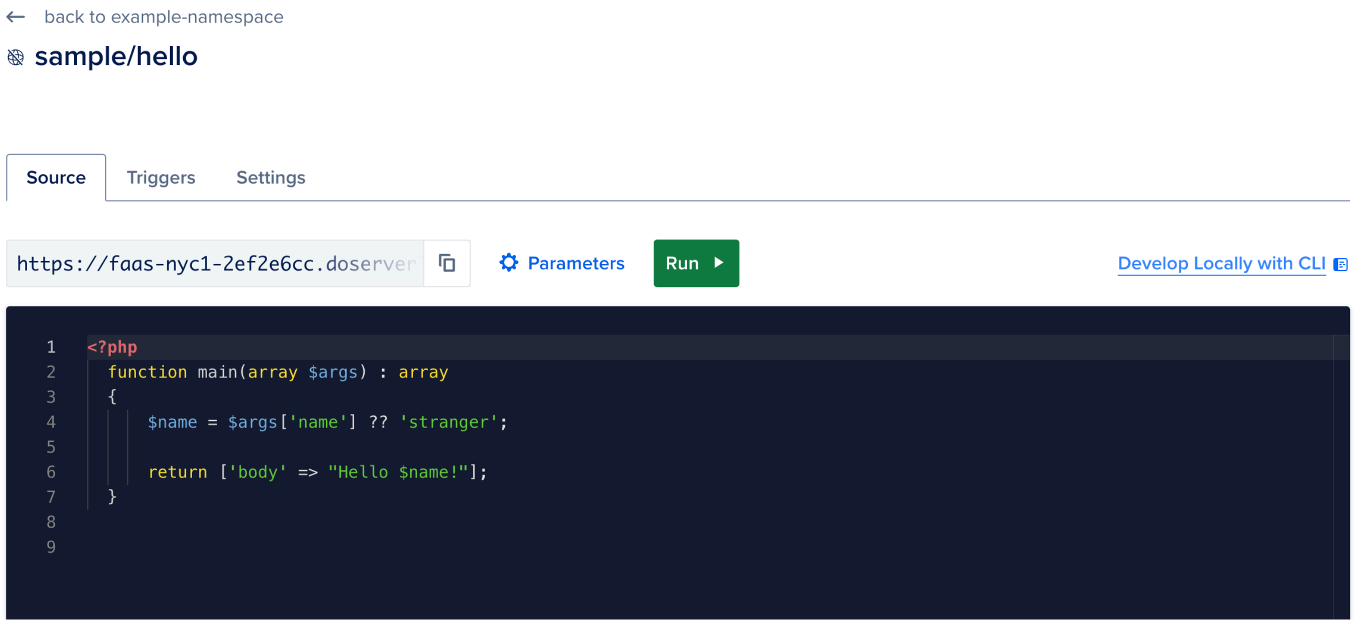
```
{
  "body": "Hello Joe!"
}
```

If you now go the UI you can see the new namespace:



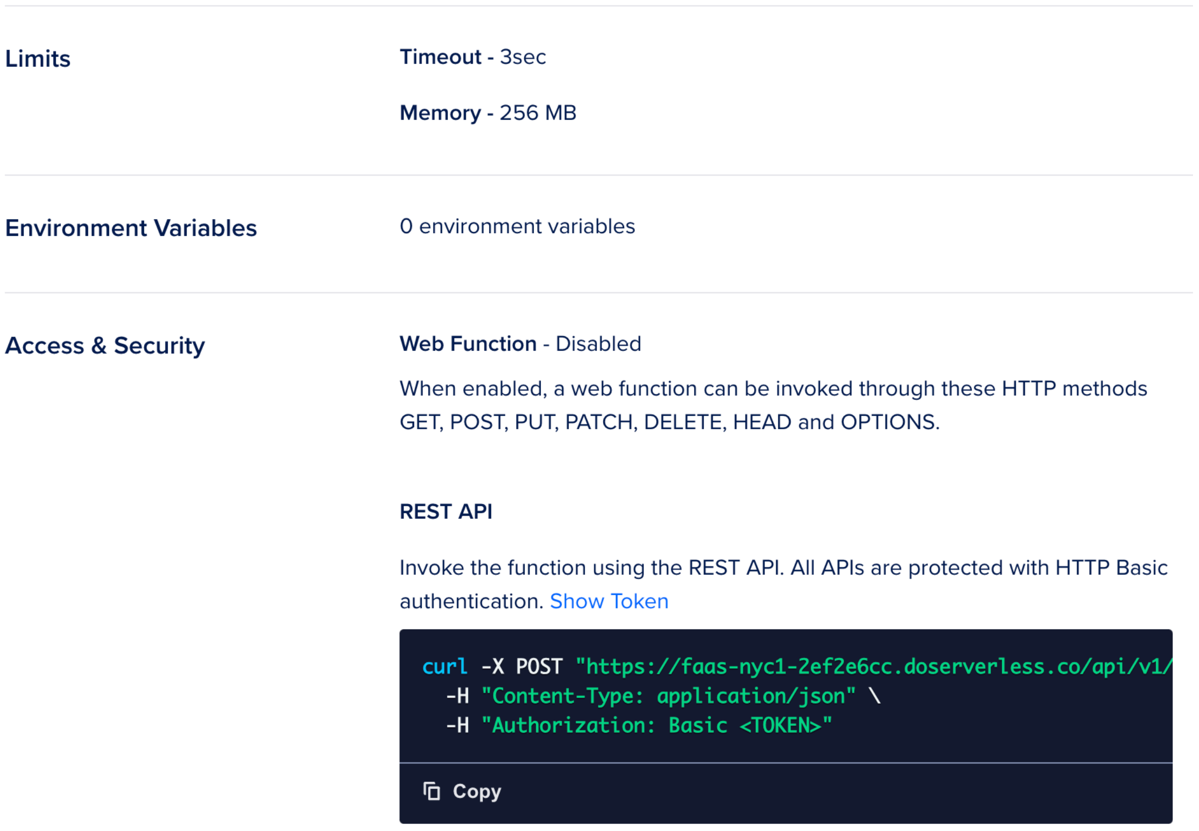
Name	Created	Region
private	7 months ago	FRA1
example-namespace	11 minutes ago	NYC1

You can also edit and test your function directly from the browser:



```php
<?php
  function main(array $args) : array
  {
      $name = $args['name'] ?? 'stranger';

      return ['body' => "Hello $name!"];
  }
```

All of this is great for development purposes. If you go the function's settings you can set access & security-related options:



Limits	**Timeout** - 3sec **Memory** - 256 MB
Environment Variables	0 environment variables
Access & Security	**Web Function** - Disabled When enabled, a web function can be invoked through these HTTP methods GET, POST, PUT, PATCH, DELETE, HEAD and OPTIONS. **REST API** Invoke the function using the REST API. All APIs are protected with HTTP Basic authentication. Show Token

```
curl -X POST "https://faas-nyc1-2ef2e6cc.doserverless.co/api/v1/
  -H "Content-Type: application/json" \
  -H "Authorization: Basic <TOKEN>"
```

Disable Web function if you don't want your function to be publicly available to anyone who knows the link. REST API provides HTTP authentication by default. So we can invoke the function such as this:

```
curl -X POST "https://faas-nyc1-2ef2e6cc.doserverless.co/api/v1/namespaces/fn-
5994b047-2ae8-4791-acc1-8997f0db7cd5/actions/sample/hello?
blocking=true&result=true" \
  -H "Content-Type: application/json" \
  -H "Authorization: Basic <TOKEN>"
```

DigitalOcean hosts the function by default on an HTTP endpoint. As you can see, `sample/hello` became part of the URL. There are two URL parameters:

- `result=true` makes your function return only your data without meta information.
- `blocking=true` means it's synchronous that returns the results immediately. If you don't provide this parameter the response is going to be this:

```
{"activationId":"a12f7ea35fea4a4eaf7ea35fea4a4ea5"}
```

You'll get only an `activationId` but not the actual results since now the function is async. You can query the results later using this activation ID.

As you can see, a serverless function is literally just one function hosted on a cloud provider without an actual server. Some advantages:

- It scales **exactly** with your incoming traffic. If you have one user hitting the create post endpoint your function runs in one instance. If there are 1000 users, then your function scales immediately. And then it scales back after the spike in traffic.
- You'll pay based on your usage. There are no fixed monthly costs.

But of course, it has disadvantages as well:

- If you really want to refactor your application into many small functions it becomes over-complicated in a pretty short time. Of course, you don't need to do that. In an upcoming chapter, we're going to deploy an entire Laravel app as a function.
- HTTP requests. Imagine your code runs for 600ms and then your it needs to send an HTTP request to a 3rd party API that takes 1200ms. 67% of your invocation time (and your money) is wasted. Your function just waits for 1.2s for an API to complete. If you call functions from other functions the situation gets even worse. Let's say function A takes 600ms to run and it calls function B which requires 400ms:
  - Function A took 1000ms
  - Function B took 400ms
  - The total execution time is 1400ms. There's an extra 40% you paid for waiting.

Imagine if function B calls the 3rd party for another 1200ms. Now function B takes 400+1200=1600, and function A takes 600+(function B's time)=600+1600=2200ms. The total execution time is 2200+1600=3800ms. Now there's an additional 2800ms that you spent waiting for stuff. And remember, you pay for these milliseconds.

The architecture of a code review tool

Overview

Now that we know the basics of functions and PaaS solutions, let's design Laracheck. Here's the application flow:

- Users register on laracheck.io
- They install the Laracheck app on their GitHub account
- They open a new pull request
- Laracheck will review the code, run its checks, and comment on the PR with the results

It needs GitHub integration. GitHub sends a `POST` request whenever a PR is opened or updated. So it seems like we need at least two components:

- **app**. This is where users can register and set up their accounts.
- **webhook**. This is the component that gets triggered by GitHub if a PR is opened. It needs to expose an API and process GitHub requests. It also needs to communicate with the app component. for example, it checks if you have a subscription or you're on a free trial, etc. And finally, it runs the code checks.

As it turned there's a really great framework for writing GitHub apps called `probot`. It's a nodejs framework. Which is great, because I love node (sorry but it's true).

So how to run the actual code checks? These checks require reading the actual source code (which was pushed to GitHub), creating an abstract syntax tree (AST) and then check it then processing the tree and looking for problems.

An AST is pretty freaking big. For example, this tiny class:

```php
<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\Facades\DB;

class PostController extends Controller
{
  public function index()
  {
    return PostResource::collection(Post::all());
  }
}
```

Results in a 426 line long JSON object.

This controller:

```php
class UninstallAppController extends Controller
{
  public function __invoke(int $installationId)
  {
    return DB::transaction(function () use ($installationId) {
      /** @var User */
      $user = User::query()
        ->whereHas('installations', function ($query) use ($installationId) {
          $query->where('installation_id', $installationId);
        })
        ->firstOrFail();

      $user->uninstallApp($installationId);

      Cache::forget("{$user->username}-repositories");

      return response()->noContent();
    });
  }
}
```

Results in a 1400 line long JSON object and it's about 25 level deep. Imagine if someone opens a PR that contains just 15 new files with 5000 lines of code. Or if it's a legacy project and someone changes 10000 lines, long classes.

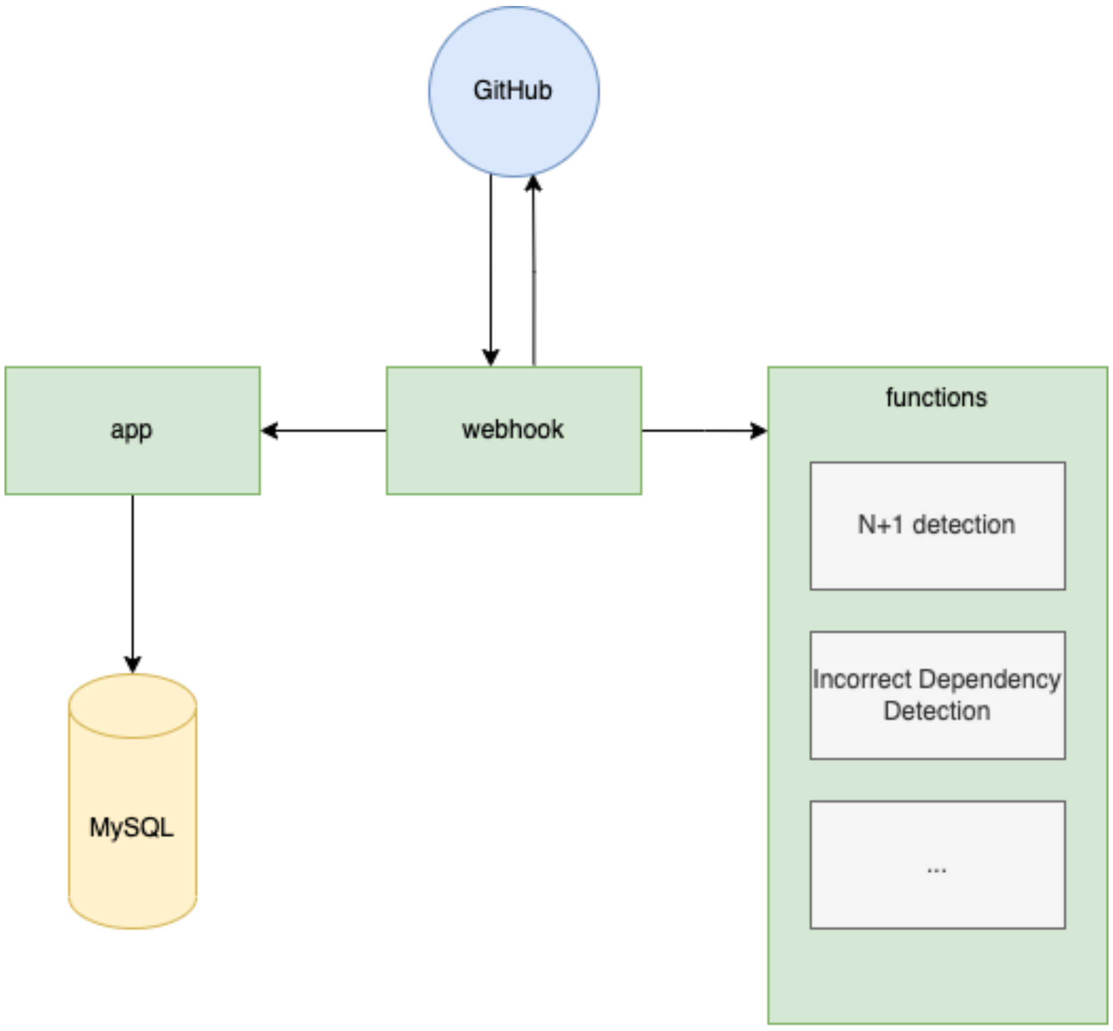
The truth is I didn't know much about these trees and code analysis at the time. I wasn't sure how resource-intensive these tasks are. I also wanted these things:

- Push-to-deploy. I push to the main branch and production is updated.
- Zero-downtime deployments.
- 1-click rollbacks. It's pretty important to me that if something goes wrong I can roll back with just one click. This comes from the lack of knowledge of code analysis in general.
- Dev/prod parity. Staging and production should be as similar as possible.

- Minimal server maintenance. Usually, I like playing around with servers and docker commands. But in this case, I wanted to focus only on the actual product.
- Easy to scale. I didn't know how much traffic to expect. I didn't know how resource-intensive is a code review tool.

For these reasons, I decided to go with functions. Each check should be its own function. It's important to note that the use case is literally perfect for an easy function setup. These code checks don't require frameworks. They can be written in literally a few files. They require only a few dependencies. No external API calls. No external storage.

This is what the architecture looks like:



- GitHub sends a request to webhook
- It gets some information from app that communicates with a MySQL database which contains user information
- It invokes the functions
- Then sends a request back to GitHub to write a new comment to the PR

This way, I think we can get the most out of everything:

- app serves the landing page and other user-facing services connected to a MySQL DB with Laravel and Blade. This is the best use case for Laravel I think.

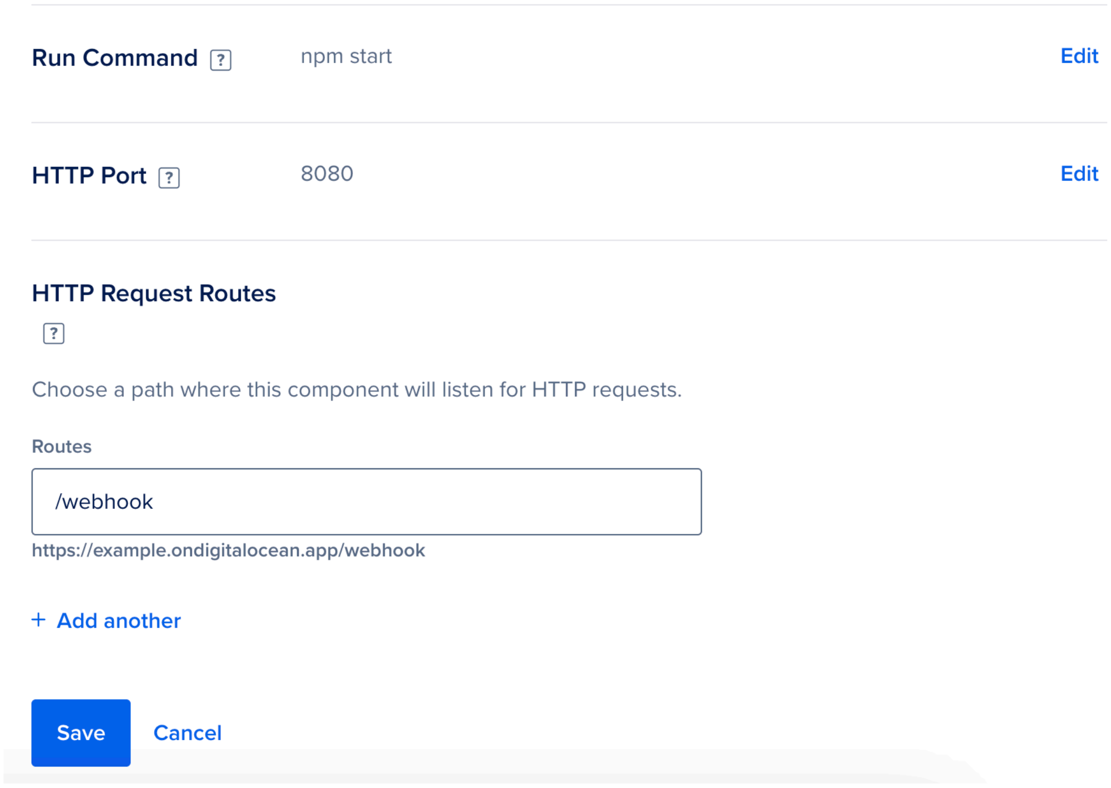
- webhook accepts HTTP requests and sends other async requests. This is literally the single best use case for a nodejs application. As you might know, nodejs is a single-threaded, async runtime engine with non-blocking I/O. It's one of the best architectures in my opinion, and it was designed for a use case such as this one. Few CPU-intensive tasks, many async I/O operations.
- And of course, functions can scale up and down on the fly as traffic grows. This eliminates all kinds of uncertainty from the project.

Webhook

Adding the webhook component to the application is almost no different from what the app was. The only difference is the build and run commands:

- Build command: `npm run build`
- Run command: `npm start`

Other than that we only change the HTTP requests route. We want the Laravel app to be accessible on `myapp.com`. Webhook needs another route:



Run Command: npm start

HTTP Port: 8080

HTTP Request Routes

Choose a path where this component will listen for HTTP requests.

Routes: /webhook

https://example.ondigitalocean.app/webhook

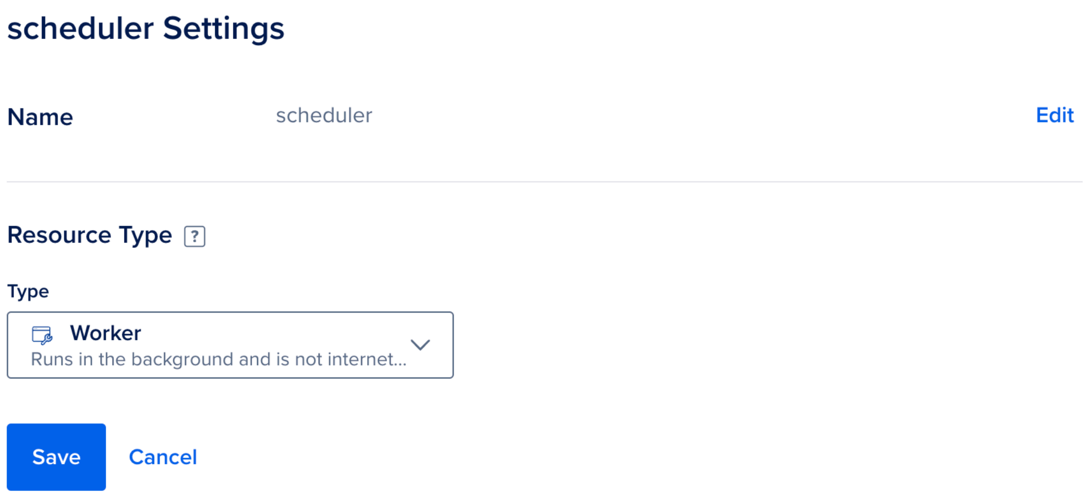
The route is `/webhook`. This means if you have a route inside the application, for example, `foo` then it can be accessed as `myapp.com/webhook/foo`. The webhook component is super lightweight so it doesn't need to be scaled.

Scheduler

The application also needs a scheduler. If you read through the whole book you already know that running a scheduler always was a bit "trickier:"

- Linux cronjob when we didn't have Docker images.
- `sleep 60 && php /usr/src/artisan schedule:run` combined with a restart policy in docker-compose.
- `php /usr/src/artisan schedule:run` combined with the more "intelligent" restart policy of Docker Swarm.
- Using the `CronJob` resource of Kubernetes with the `* * * * *` schedule setting.

The App Platform is no exception. Adding a scheduler means adding the main repo as a `worker` component:


scheduler Settings

Name scheduler

Resource Type

Type: Worker — Runs in the background and is not internet...

A worker is a kind of component that runs in the background and is not exposed to the outside world. We also need to change the run script to this:

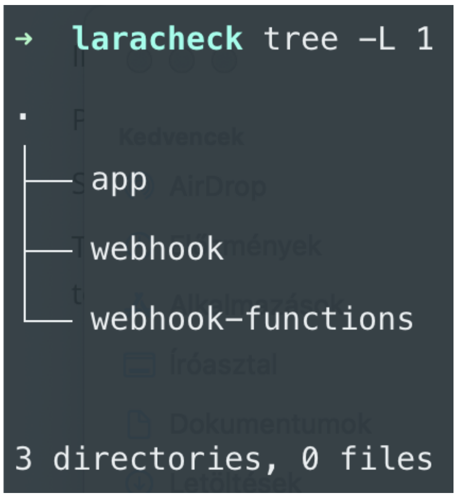
```
while true; do
  php artisan schedule:run || true;
  sleep 60;
done
```

If you want to add a queue worker you need to do the same but the script should be:

```
php artisan queue:work
```

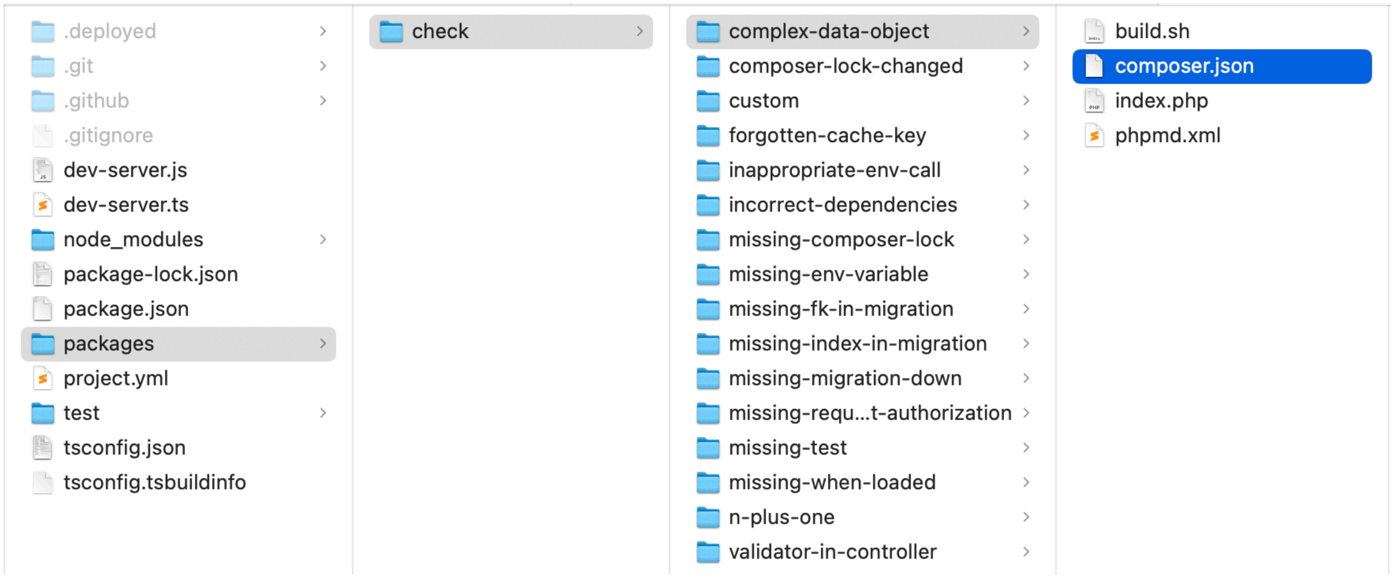
Adding functions

The last step is to add the functions. In DigitalOcean, we can attach functions directly to an app just like any other component. So I created a GitHub repo for them. By the way, I also use a monorepo just as I did with the sample project in the book:



```
→ laracheck tree -L 1
.
├── app
├── webhook
└── webhook-functions

3 directories, 0 files
```

This is the exact structure of the functions repo:

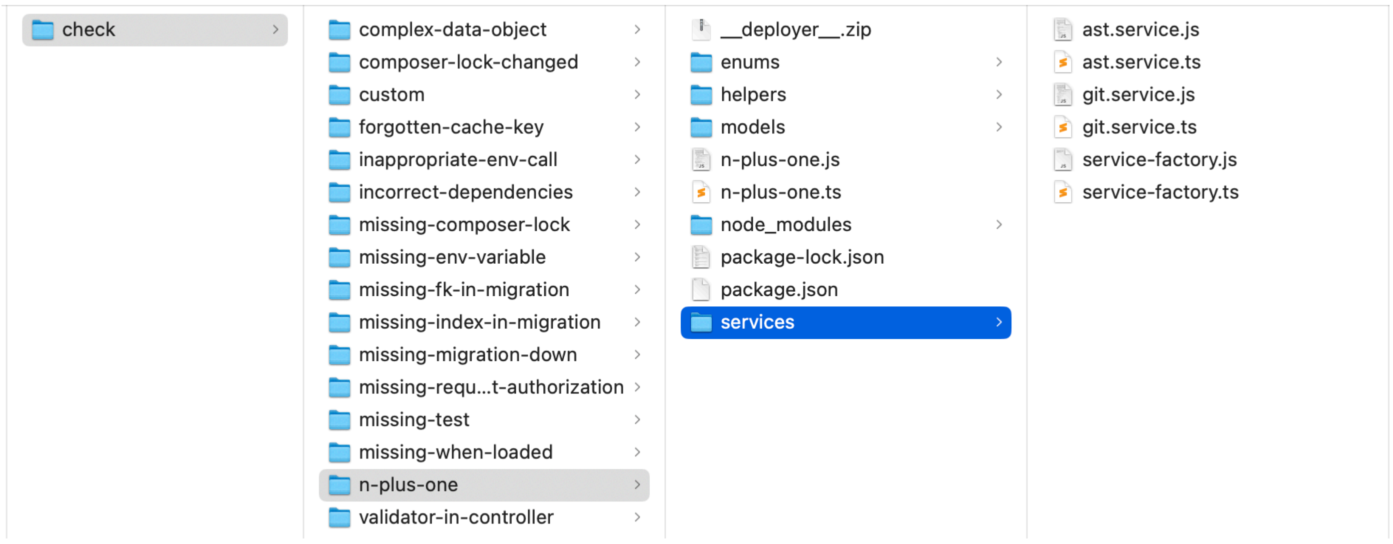


```
.deployed          check    complex-data-object                 build.sh
.git                        composer-lock-changed               composer.json
.github                     custom                              index.php
.gitignore                  forgotten-cache-key                 phpmd.xml
dev-server.js               inappropriate-env-call
dev-server.ts               incorrect-dependencies
node_modules                missing-composer-lock
package-lock.json           missing-env-variable
package.json                missing-fk-in-migration
packages                    missing-index-in-migration
project.yml                 missing-migration-down
test                        missing-requ...t-authorization
tsconfig.json               missing-test
tsconfig.tsbuildinfo        missing-when-loaded
                            n-plus-one
                            validator-in-controller
```

I have only one package called `check` and each function is a directory. Each function can have its own dependencies. For example, this is the `composer.json` of the `complex-data-object` function:

```json
{
  "name": "laracheck/complex-data-object",
  "autoload": {
      "psr-4": {
          "": "./"
      }
  },
  "authors": [{
      "name": "Martin Joo",
      "email": "martin@laracheck.io"
  }],
  "require": {
      "php": "8.0.*",
      "phpmd/phpmd": "^2.13"
  }
}
```

I also have nodejs services, for example `n-plus-one`:

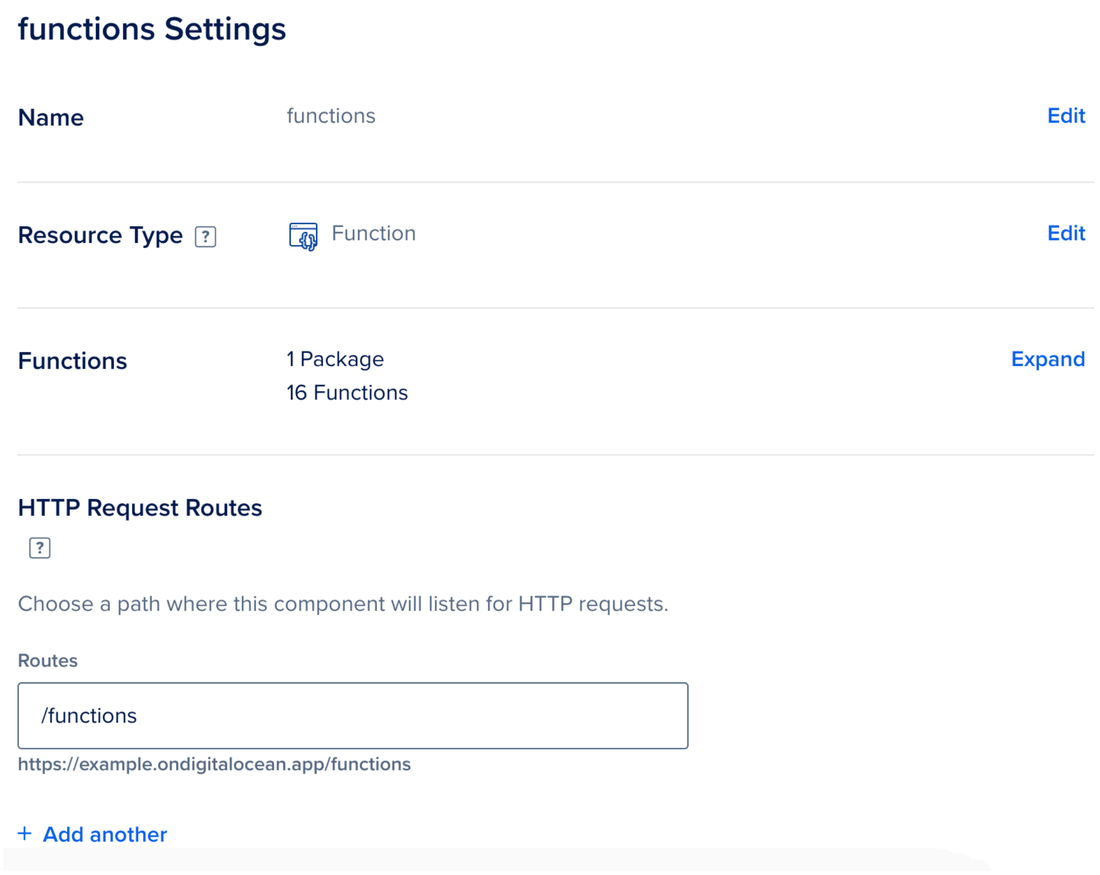


This is another **advantage** of using functions. You can use multiple languages in your project easily. Why I write a PHP code analysis tool in nodejs is another question. But the short answer is: believe it or not, I'm much better at writing low-level code using nodejs because I was a node developer before I moved to Laravel (sorry about that).

To call these functions we need to use HTTP requests from the webhook. To simplify things I created a class called `FunctionClient`:

```
import axios from 'axios';
import Rollbar from 'rollbar';
import { config } from '../config/app.config';

export default class FunctionClient {
  rollbar: Rollbar;

  constructor(rollbar: Rollbar) {
    this.rollbar = rollbar;
  }

  async invoke(functionName: string, args: any): Promise<boolean> {
    try {
      const { data } = await axios.post(
        `${config.functions.host}/check/${functionName}?
blocking=true&result=true`, args,
        {
          headers: {
            'Authorization': `Basic ${config.functions.token}`,
            'Content-Type': 'application/json',
          },
      });

      return data.result;
    } catch (err: any) {
      this.rollbar.critical(`Function error: ${functionName}`, {
        function: functionName,
        args,
        error: err,
      });

      return true;
    }
  }
}
```

You can add functions as a new component to your existing app. There's a function resource type:



functions Settings

Name	functions	Edit
Resource Type	Function	Edit
Functions	1 Package 16 Functions	Expand

HTTP Request Routes

Choose a path where this component will listen for HTTP requests.

Routes

/functions

https://example.ondigitalocean.app/functions

It has another route so we can call the function such as this: `myapp.com/functions/check/n-plus-one` where `functions` is the HTTP request route, `check` is the package name in the repository, and `n-plus-one` is one of the functions.

Serverless Laravel on AWS

Serverless functions seem exciting. But when I learned about them, I had only one question: "All right. So we have these small files. Great. But how the hell can I combine Laravel with functions? Do I need to drop Laravel and write bare-bone PHP code if I want a function? I won't do that."

Functions were originally designed to run really really small services. Such as we did in the previous chapter. However, in a bigger application, it introduces lots of complexity. Just imagine a bigger Laravel project you worked on. Let's say it has 100 API endpoints. Does it mean you need to refactor your monolith app to 50+ services backed by 50+ databases? Fortunately, no. It introduces problems you didn't even know existed before. People won't do that but they do want to enjoy the convenience and "infinite-scale" of functions. As it turned out we can easily deploy a whole Laravel app to a function.

In this chapter, we're going to deploy a whole Laravel app to AWS Lambda functions. We need to use two libraries:

- serverless
- bref

serverless is a framework that helps you develop and deploy AWS lambda functions and other AWS resources your application might need. It's a CLI tool built on top of the AWS CLI

The serverless framework

Before working with Laravel, let's just deploy a simple PHP file to AWS Lambda using the serverless framework. It's a framework built on top of AWS Cloudformation. Cloudformation is an infrastructure-as-code tool that lets you write the required AWS resources in a YAML or JSON file and then you can provision them. Serverless is a simplified version of that that enabled us to use pretty simple YAML files to describe the infra and then it uses Cloudformation under the hood to provision it.

To install it you need to run:

```
npm install -g serverless
```

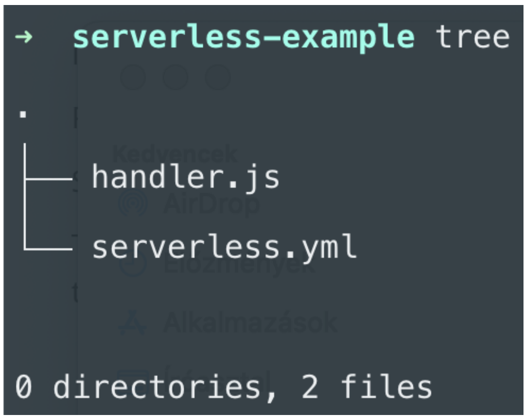
Before you start, create an IAM user on AWS or **only for development purposes** use your root user's keys:

```
serverless config credentials --provider aws --key <key> --secret <secret>
```

Here's how you can create a new project:

```
serverless create --template aws-nodejs --path serverless-example
```

The `template` option defines what files and config it should create initially. It's just a quick nodejs example and we're going to use Laravel again:



```
→  serverless-example tree
.
├── handler.js
└── serverless.yml

0 directories, 2 files
```

It provides us with a sample js file:

```
'use strict';

module.exports.hello = async (event) => {
  return {
    statusCode: 200,
    body: JSON.stringify(
      {
        message: 'Hello world',
        input: event,
      },
      null,
      2
    ),
  };
};
```

It also generates a `serverless.yaml` which is the main configuration for the serverless framework:

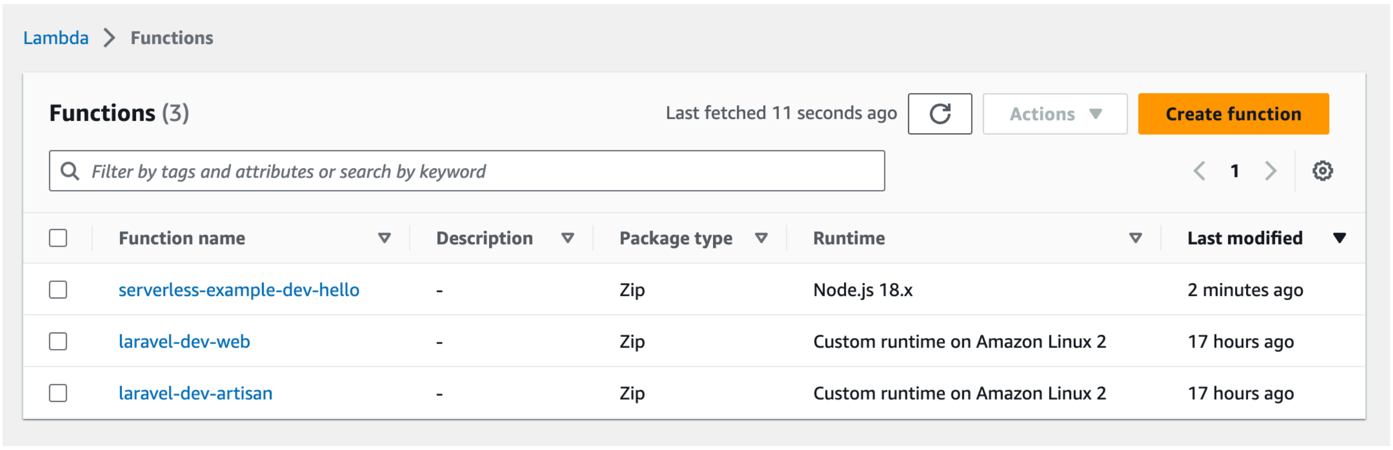
```
service: serverless-example
frameworkVersion: "3"

provider:
  name: aws
  runtime: nodejs18.x

functions:
  hello:
    handler: handler.hello
```

The most important thing about this file is the `functions` section. It defines a `hello` function with a `handler` of `handler.hello`. it refers to the `handler.js` file and the `hello` function inside it.

And now we can deploy the function:

```
serverless deploy
```

After the function is deployed you should be able to see it on AWS console:


Lambda > Functions
Functions (3)
Last fetched 11 seconds ago
Actions
Create function
Filter by tags and attributes or search by keyword
1

Function name	Description	Package type	Runtime	Last modified
serverless-example-dev-hello	-	Zip	Node.js 18.x	2 minutes ago
laravel-dev-web	-	Zip	Custom runtime on Amazon Linux 2	17 hours ago
laravel-dev-artisan	-	Zip	Custom runtime on Amazon Linux 2	17 hours ago

The first is the one I just deployed.

You can invoke the function using this command:

```
serverless invoke -f hello
```

It should the return this json:

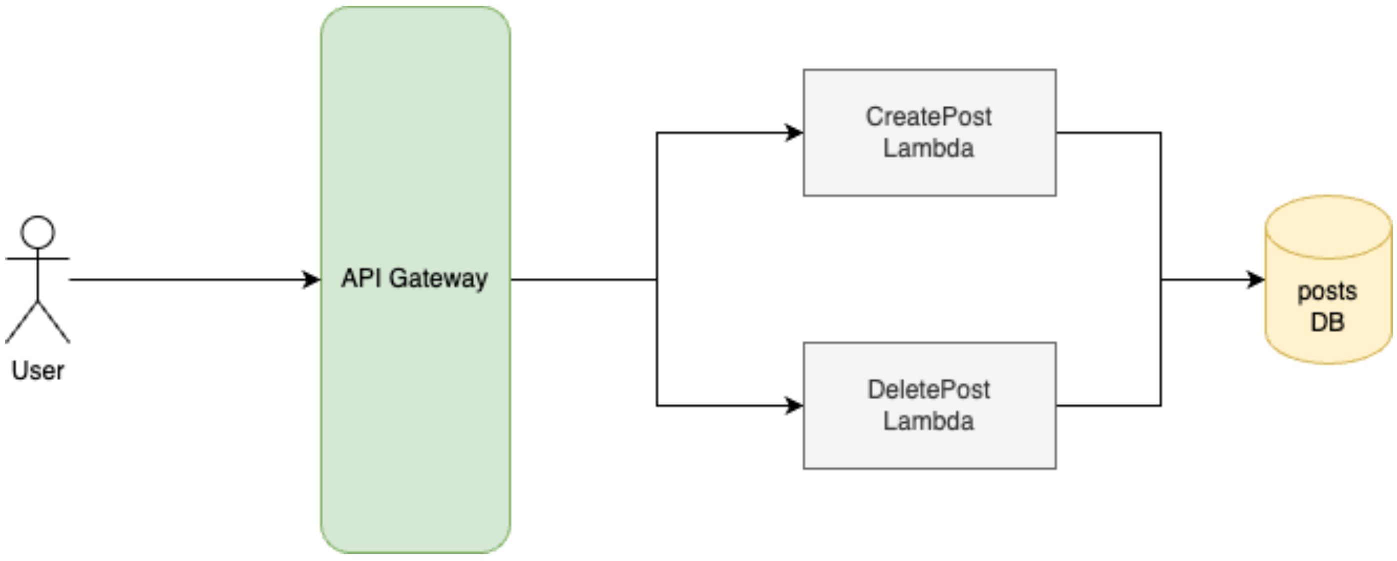
```
{
  "statusCode": 200,
  "body": {"message": "Hello world", "input": {}}
}
```

It's all great but not very useful yet. Users won't use the terminal to invoke our functions. And also, for just deploying a simple function we don't really need a framework. AWS CLI can do that. DigitalOcean CLI can do that. The power of `serverless` comes from when you want to combine things. When you want to use more infrastructure than just a simple function.

So let's expose the function via an API Gateway. An AWS API gateway is pretty similar to:

- An nginx reverse proxy we used with Docker
- An ingress we used with Kubernetes

It exposes an API to the outside world and then forwards the requests to the right services. In this case, a service is a Lambda function:



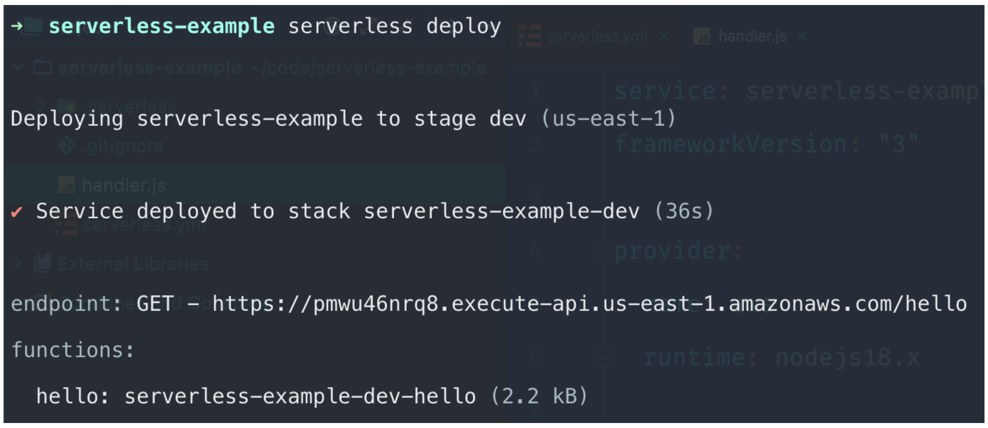
So `serverless` enables us to create an API gateway from the configuration. It's basically infrastructure-as-code:

```yaml
functions:
  hello:
    handler: handler.hello
    events:
      - httpApi:
          path: /hello
          method: get
```

I added an `events` section to the function. This means:

- There's an `event` that will invoke the Lambda
- The event is an HTTP request to the `/hello` endpoint

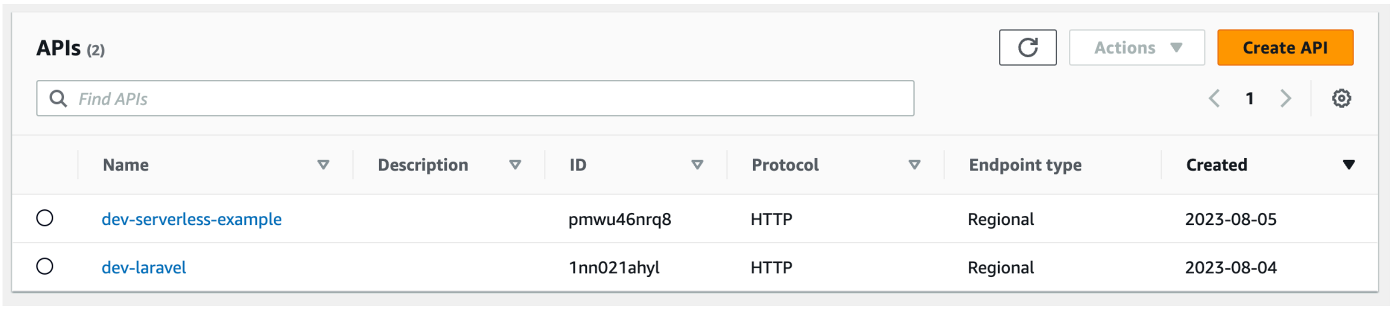
If we now re-run `serverless deploy` we'll get a message such as this:



```
➜  serverless-example serverless deploy

Deploying serverless-example to stage dev (us-east-1)

✔ Service deployed to stack serverless-example-dev (36s)

endpoint: GET - https://pmwu46nrq8.execute-api.us-east-1.amazonaws.com/hello

functions:
  hello: serverless-example-dev-hello (2.2 kB)
```

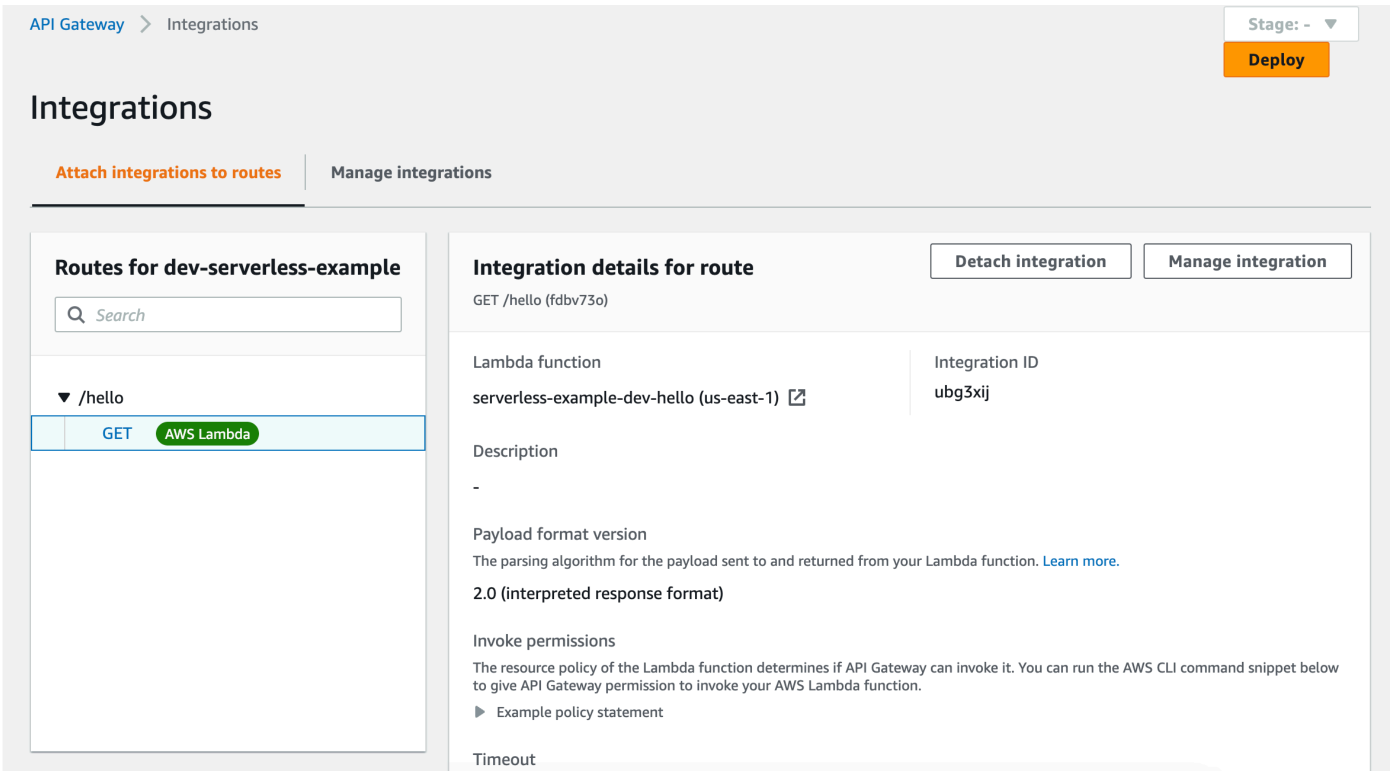
There's a URL for the function! Mine is this: https://pmwu46nrq8.execute-api.us-east-1.amazonaws.com/hello

In the AWS console you can see there's a new API gateway:



	Name	Description	ID	Protocol	Endpoint type	Created
○	dev-serverless-example		pmwu46nrq8	HTTP	Regional	2023-08-05
○	dev-laravel		1nn021ahyl	HTTP	Regional	2023-08-04

In the API gateway, there's a panel called `Integration`. Here you can see `serverless` successfully attached the lambda to the HTTP endpoint:



Integration details for route

GET /hello (fdbv73o)

Lambda function

serverless-example-dev-hello (us-east-1)

Integration ID

ubg3xij

Description

-

Payload format version

The parsing algorithm for the payload sent to and returned from your Lambda function. Learn more.

2.0 (interpreted response format)

Invoke permissions

The resource policy of the Lambda function determines if API Gateway can invoke it. You can run the AWS CLI command snippet below to give API Gateway permission to invoke your AWS Lambda function.

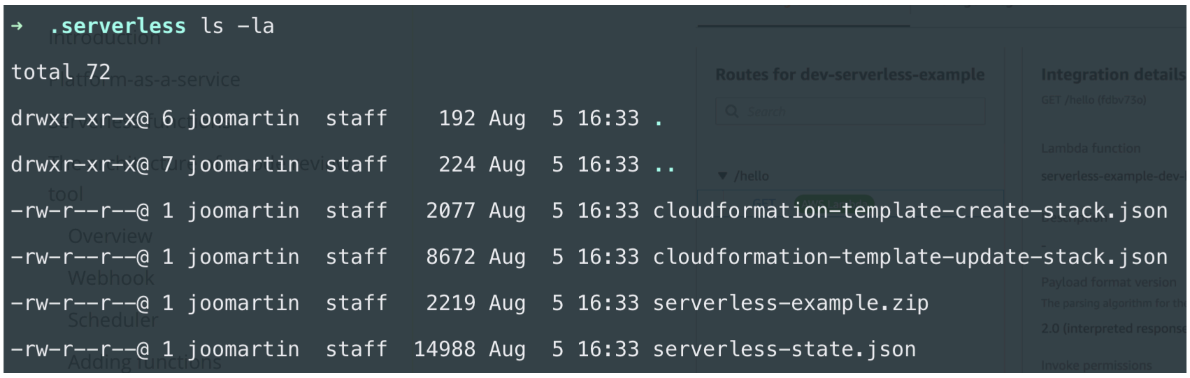
Timeout

These are just the basics of `serverless`. It can create so many things for your application:

- Websockets
- Schedulers
- Event streams
- CloudWatch logging
- S3 storage

It's all configuration-driven which is a pretty good thing. Your repository contains the whole infrastructure that your application needs.

If you run `serverless deploy` it creates a `.serverless` folder. Inside that folder, you can see your project as a ZIP file, and the Cloudformation config generated by serverless:



```
➜  .serverless ls -la
total 72
drwxr-xr-x@ 6 joomartin  staff    192 Aug  5 16:33 .
drwxr-xr-x@ 7 joomartin  staff    224 Aug  5 16:33 ..
-rw-r--r--@ 1 joomartin  staff   2077 Aug  5 16:33 cloudformation-template-create-stack.json
-rw-r--r--@ 1 joomartin  staff   8672 Aug  5 16:33 cloudformation-template-update-stack.json
-rw-r--r--@ 1 joomartin  staff   2219 Aug  5 16:33 serverless-example.zip
-rw-r--r--@ 1 joomartin  staff  14988 Aug  5 16:33 serverless-state.json
```

bref

The reason I used nodejs in the previous chapter is that AWS Lambda doesn't have a PHP runtime. It can be solved with 3rd party tools and AWS layers, however.

Bref is a composer package that provides php runtime on AWS, deployment tooling for PHP applications, and excellent Laravel integrations. It works together with `serverless`.

If you just quickly want to test it with a single `index.php` you can do it in 2 minutes:

```
mkdir bref-test
cd bref-test
composer require bref/bref
vendor/bin/bref init
serverless deploy
```

But let's try it with a brand new Laravel installation:

```
composer create-project laravel/laravel serverless-laravel
```

Then:

```
composer require bref/bref
```

After it's installed, publish the `serverless.yml` config file:

```
php artisan vendor:publish --tag=serverless-config
```

Let's look at the generated config:

```
service: laravel

provider:
    name: aws
    region: us-east-1
    environment:
        APP_ENV: production
```

```yaml
package:
    patterns:
        - '!node_modules/**'
        - '!public/storage'
        - '!resources/assets/**'
        - '!storage/**'
        - '!tests/**'

functions:
    web:
        handler: public/index.php
        runtime: php-81-fpm
        timeout: 28
        events:
            - httpApi: '*'

    artisan:
        handler: artisan
        runtime: php-81-console
        timeout: 720

plugins:
    - ./vendor/bref/bref
```

The `provider` is pretty similar to the earlier configs. It now contains an `environment` section. When you deploy with `bref` your `.env` file is going to be deployed as well, but here you can override values. `package.patterns` excludes files from the deployment.

The file defines two functions:

- `web` is going to be the entire application. As you can see, it listens to every HTTP event. It means you get the frontend by visiting `/` and accessing the API by hitting an URL such as `/api/users`. The runtime `php-81-fpm` comes from the `bref` the plugin. This is what I talked about earlier. `bref` provides PHP runtimes on AWS. It uses `fpm` just as we did in the book so everything should work the same.
- `artisan` is a function that runs on the `php-81-console` runtime which is a CLI PHP runtime and it enables us to run `artisan` commands. It's similar to the official PHP CLI image on DockerHub.

There's a `BrefServiceProvider` that makes the application "lambda-compatible" by default, for example:

```
protected function fixDefaultConfiguration()
{
  if (Config::get('session.driver') === 'file') {
    Config::set('session.driver', 'cookie');
  }

  if (Config::get('logging.default') === 'stack') {
    Config::set('logging.default', 'stderr');
  }
}
```

Just as I said in the book (multiple times I guess) you cannot have settings such as `file` driver in a cloud environment. Lambda won't let you log into files either, so the `stack` driver is out of the picture as well.

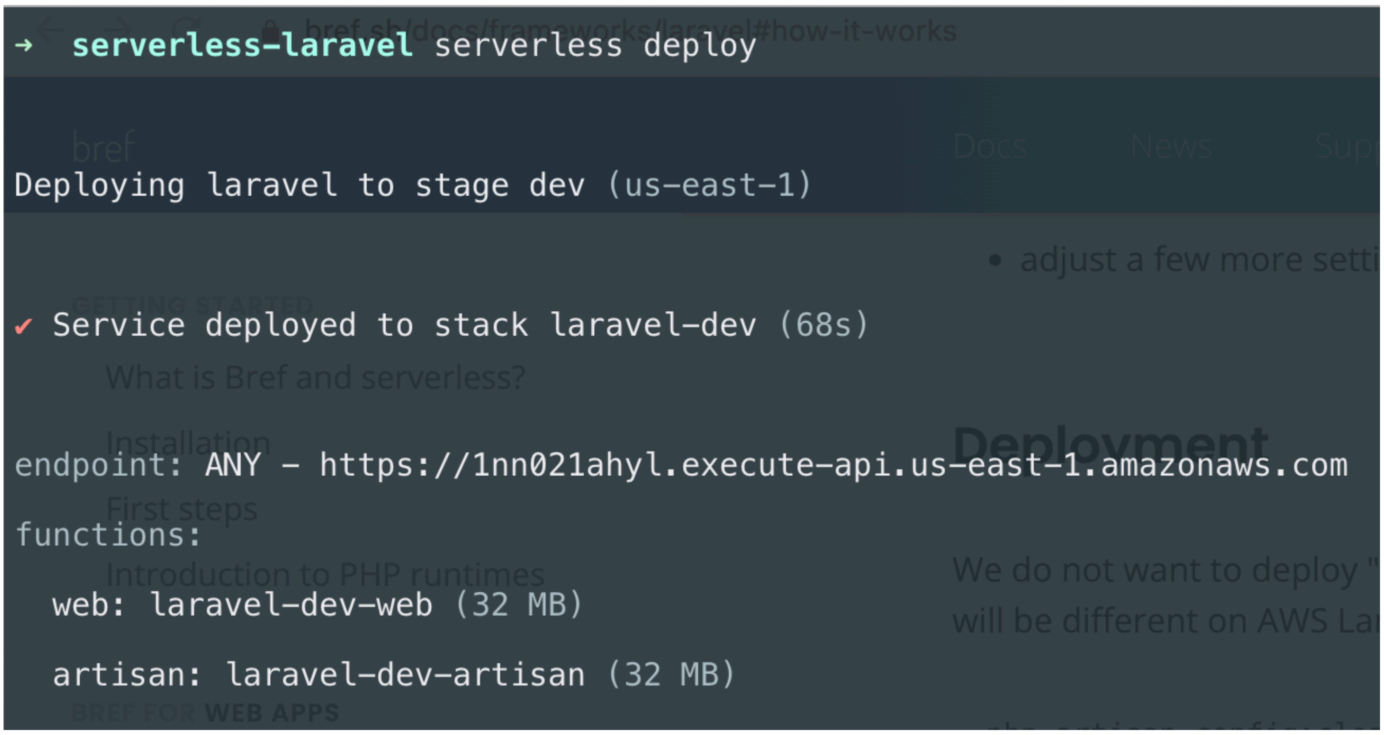
Before deploying with `bref` you need to clear the config cache:

```
php artisan config:clear
```

Then just run:

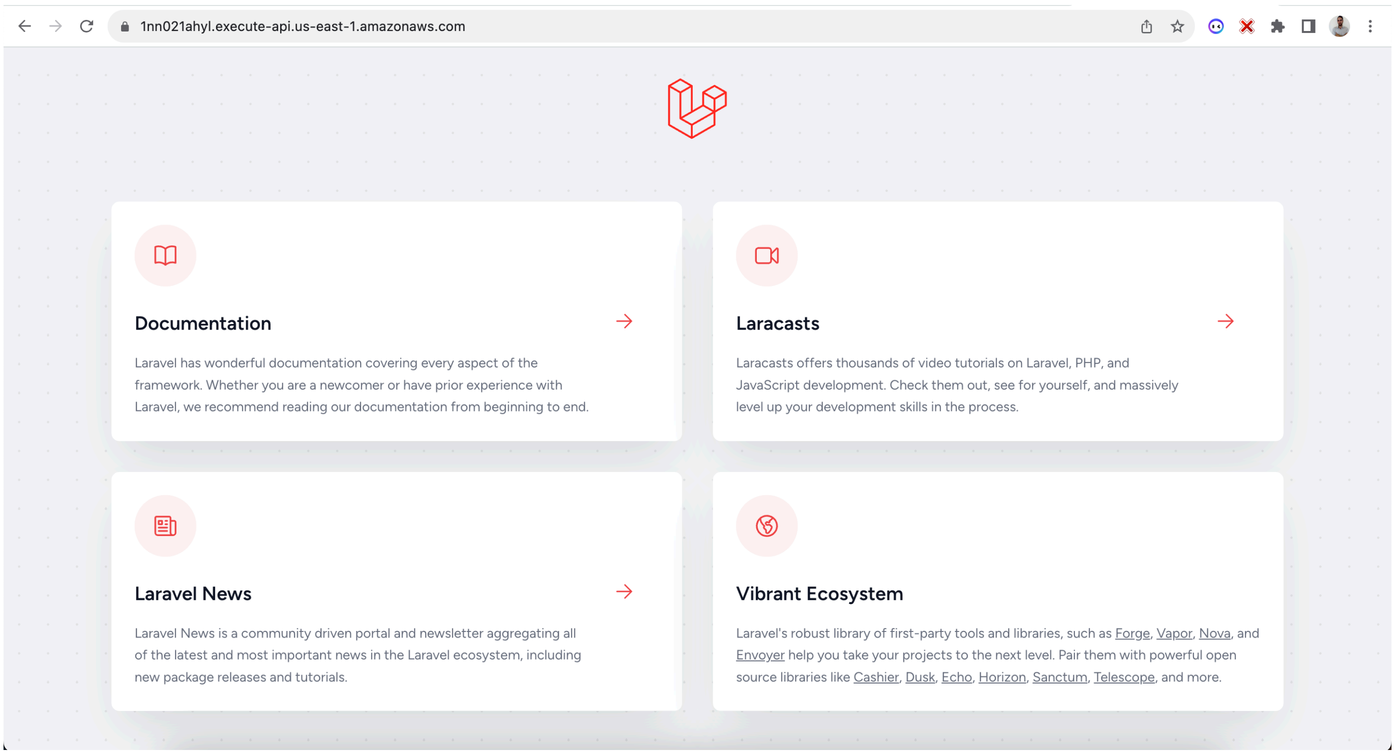
```
serverless deploy
```

The output should be something like this:



```
→ serverless-laravel serverless deploy

Deploying laravel to stage dev (us-east-1)

✔ Service deployed to stack laravel-dev (68s)

endpoint: ANY - https://1nn021ahyl.execute-api.us-east-1.amazonaws.com
functions:
  web: laravel-dev-web (32 MB)
  artisan: laravel-dev-artisan (32 MB)
```

If you visit the URL given by command, you should see a fresh Laravel installation:



1nn021ahyl.execute-api.us-east-1.amazonaws.com

Documentation

Laravel has wonderful documentation covering every aspect of the framework. Whether you are a newcomer or have prior experience with Laravel, we recommend reading our documentation from beginning to end.

Laracasts

Laracasts offers thousands of video tutorials on Laravel, PHP, and JavaScript development. Check them out, see for yourself, and massively level up your development skills in the process.

Laravel News

Laravel News is a community driven portal and newsletter aggregating all of the latest and most important news in the Laravel ecosystem, including new package releases and tutorials.

Vibrant Ecosystem

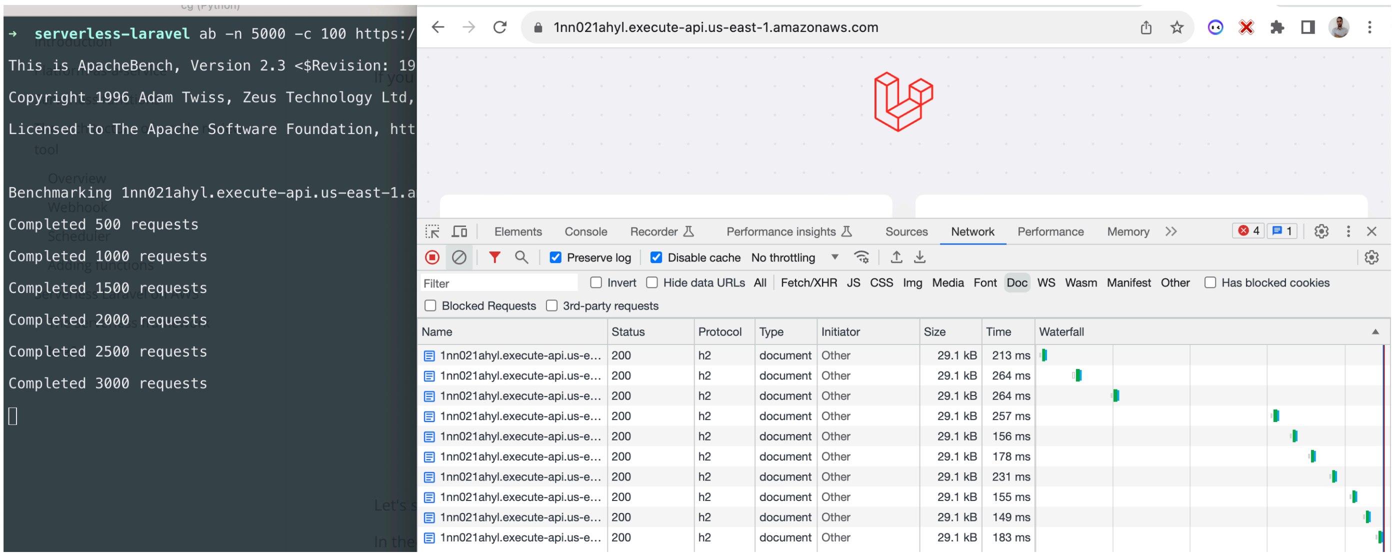
Laravel's robust library of first-party tools and libraries, such as Forge, Vapor, Nova, and Envoyer help you take your projects to the next level. Pair them with powerful open source libraries like Cashier, Dusk, Echo, Horizon, Sanctum, Telescope, and more.

Let's summarize what just happened: by running **4 commands** we deployed a new Laravel app to Amazon Lambda.

In theory, the site should scale as traffic grows, right? Let's test it:

```
ab -n 5000 -c 100 https://1nn021ahyl.execute-api.us-east-1.amazonaws.com/
```

This sends 5000 requests to the sites 100 of which are concurrent. As the test was going, I was able to get the site under **300ms**:



```
→ serverless-laravel ab -n 5000 -c 100 https:/
This is ApacheBench, Version 2.3 <$Revision: 19
Copyright 1996 Adam Twiss, Zeus Technology Ltd,
Licensed to The Apache Software Foundation, htt

Benchmarking 1nn021ahyl.execute-api.us-east-1.a
Completed 500 requests
Completed 1000 requests
Completed 1500 requests
Completed 2000 requests
Completed 2500 requests
Completed 3000 requests
```

1nn021ahyl.execute-api.us-east-1.amazonaws.com
Elements Console Recorder Performance insights Sources Network Performance Memory
Preserve log Disable cache No throttling
Filter Invert Hide data URLs All Fetch/XHR JS CSS Img Media Font Doc WS Wasm Manifest Other Has blocked cookies
Blocked Requests 3rd-party requests

Name	Status	Protocol	Type	Initiator	Size	Time	Waterfall
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	213 ms	
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	264 ms	
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	264 ms	
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	257 ms	
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	156 ms	
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	178 ms	
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	231 ms	
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	155 ms	
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	149 ms	
1nn021ahyl.execute-api.us-e...	200	h2	document	Other	29.1 kB	183 ms	

It was able to scale up to 92.29 requests/second instantly. 100% of requests were successful. So yeah, it can scale as traffic grows.

To use the `artisan` function you need to run:

```
serverless bref:cli --args="about"
```

It runs `php artisan about` which gives you information about your app.

Check out bref's documentation. It's an excellent package, try it out. Lots of opportunities in AWS, serverless, and bref.

Conclusions

Functions are awesome, PaaS is awesome, but the question is: when should you use them?

If you're an indie hacker who wants to deploy a project fast and doesn't want to care about infrastructure, just pick your favorite PaaS/function provider and ship your stuff. It's the most easy/fast choice you have. You can always switch later if it turns out to be too expensive, not flexible enough, etc. If you have one application you can probably change your deployment strategy in a weekend or so. But these solutions probably provide the best time-to-market.

Thank you very much for reading this book! If you liked it, don't forget to Tweet about it. If you have questions you can reach out to me here.