www.phparch.com

May 2019
Volume 18 - Issue 5

php[architect]

Serverless, ReactPHP, and Expanding Frontiers

Deploying ReactPHP Applications

Serverless PHP With Bref, Part One

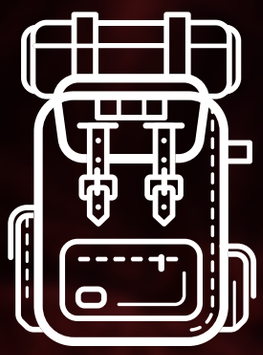
MySQL 8.0 Geographic Information System or How Did I Get to This Point?

Department of Breaking Changes: Launching PHP 7 in a Highly Available Web World

Free Sample Article

ALSO INSIDE

DAYCAMP 4 DEVELOPERS



DATA

MAY 31ST • ONLINE

DATA IS MORE VALUABLE THAN OIL.
DATA IS THE LIFEBLOOD OF MORE AND MORE COMPANIES EVERY DAY.

In every enterprise, data comes from multiple inputs and must be harnessed and combined to become actionable. If you don't tame and act on your data streams, you lose value as it flows through your systems and you miss opportunities. The decisions for dealing with your data go beyond how you store, search, and share it with others throughout your organization.

Join us online for the next Day Camp 4 Developers and learn to tame your data. Turn it from a wandering stream to a powerful tool that can inform your enterprise.

MAY 31, 2019 | 9:00 AM – 3:00 PM CDT | ONLINE

DAYCAMP4DEVELOPERS.COM

Tickets include attendance to the live event as well as access to download the videos within seven days after the event.

Serverless PHP With Bref, Part One

Rob Allen

In recent years, a different way to build applications has arisen called *serverless computing*. This term is not a good name; I'm reminded of the adage that there are two hard problems in programming: naming things, cache invalidation, and off-by-one errors. Serverless computing as a name implies no servers, but there are servers—you don't think about them though. A better way to think about it is that the servers and the management of them are abstracted away and the provider manages the allocation of resources, not the application architect.

Amazon Lambda[1] arguably started Serverless computing. It is a system where your code runs on demand and is not even in memory if it's not running. We call this *functions as a service* (FaaS). Modular code executes in response to an event, and so different parts of our application can be scaled independently without any additional effort on our part. The main advantages are that we spend more our time concentrating on the core of an application that brings value and far less worrying about how to run and scale it. There's also a lovely side-benefit in that you only pay when your code is running, so your costs are proportional to usage. As FaaS encourages building small independent units of functionality, you also get modular applications.

Of course, there were also disadvantages. It is harder to reason about an application which is spread over many separate functions and services. It's also harder to debug and keep track of which functions are deployed where. FaaS and serverless are not a solution to every problem but are a good solution for a large number of use-cases.

In this article, I'm going to walk through how to use Lambda with PHP to execute a function. In part two, I will build a static website hosted on S3, with a CDN updated via a Lambda function. To do this, we use Bref[2]. Created by Matthieu Napoli, Bref contains all the functionality required to deploy PHP functions into Lambda in a simple, clean and easy manner. It provides runtimes for PHP 7.2 and 7.3. It is closely tied to Amazon, so if you're interested in running PHP in say IBM Cloud Functions, you would use a different tool, such as Serverless Framework[3].

Let's get started by setting up AWS.

Setting Up AWS

Let's start by creating a single Lambda *Hello World* application. First, we have to start with our prerequisites: the AWS command line tools and an AWS account. This involves more steps than you would initially think because AWS has a robust and complete permissions system called IAM[4] that controls user access, so in addition to installing the command line tools, we also need to set up a user with enough permissions to create our application.

There are two command line tools we need: `aws` and `sam`. As you can guess, from its name, `aws` is the command line tool which allows access to all of the AWS system. The other tool, `sam` is the way we interact with AWS Serverless Application Model (SAM) system which is used by Bref. To install the `aws` command line tool, head to https://aws.amazon.com/cli/ and follow the instructions[5]; for `sam`.

Now we have the tools we need to set up credentials so SAM can create and manage the application resources for us. If you don't have an AWS account, head to https://console.aws.amazon.com and create one. The command line tools require access keys. To do this, we need a new user as we don't want to use our master user. This process is well-described in the Serverless Framework documentation[6].

In summary:

1. Create or log into your AWS account and go to the *Identity and Access Management* (IAM) page.
2. Click on **Users** then **Add User** and enter the username `bref-agent`.
3. For the *Access type*, select **Programmatic access** only as this user needs CLI and API access, but not web console access.
4. Click **Next: Permissions** to set permissions for our user.
5. Click **Attach existing policies directly** and then *Create Policy*.
6. In the new tab that opens, click **JSON** and paste in the JSON from Listing 1.
7. Click *Review Policy* and assign it a name of "bref-agent-policy," then click **Create Policy**, and close the tab to return to the tab where we are adding our user.
8. Click the refresh button (two arrows in a circle) on the right to refresh the list of available policies

1 *Amazon Lambda:* https://aws.amazon.com/lambda/

2 *Bref:* https://bref.sh

3 *Serverless Framework:* https://serverless.com

4 *IAM:* https://aws.amazon.com/iam/

5 *the instructions:* https://phpa.me/aws-serverless-sam

6 *Serverless Framework documentation:* https://phpa.me/aws-serverless-credentials

9. Find *bref-agent-policy* in the list, check the box next to it, and click **Next: Tags**.
10. We don't want to add any tags, so click *Next: Review*, and then **Create user**.

Listing 1 is the set of permissions we give to the user that creates our serverless application, so it has administrative permissions to create S3 buckets, DynamoDB tables, CloudFormation stacks, IAM policies, and so on. It is a fairly open policy so for production use; you may want to lock it down some more with specific permissions for each group.

We now have a new user called `bref-agent`. Note the *Access key ID* and the *Secret access key*.

Configure the AWS CLI

The easiest way to configure the `aws` and `sam` tools is to run `aws configure`.

```
$ aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/EXAMPLEKEY
Default region name [None]: us-east-1
Default output format [None]: json
```

Use your *Access key ID* and the *Secret access key* at the prompts, set your region to `us-east-1`, and use JSON for your output format.

Now that we have AWS set up, we can write our first Lambda function.

Hello World With Bref

To create a Lambda function using Bref, we start with a new directory and install Bref into it:

```
$ mkdir bref-hello-world
$ cd bref-hello-world
$ composer require mnapoli/bref
```

These commands installed the Bref code into the `vendor/` folder, so we can go ahead a create the project now. You'll see something like Output 1

Bref supports three different kinds of serverless projects depending on your needs. For our case, we want a standard Lambda function, so we select the `[0] PHP function` option, and then Bref creates our files. To learn more about the options, see the Bref documentation on runtimes[7]

What Files Do We Have?

`bref init` has created two files of interest: `template.yaml` and `index.php`. Starting with `template.yaml`, we have the SAM template which is the configuration file which defines our application and in our case contains the definition of our first Lambda function (see Listing 2).

7 *runtimes:* https://bref.sh/docs/runtimes/

Listing 1

```
{
    "Statement": [
        {
            "Action": [
                "apigateway:*",
                "cloudformation:*",
                "dynamodb:*",
                "events:*",
                "iam:*",
                "lambda:*",
                "logs:*",
                "s3:*",
                "sns:*",
                "states:*"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ],
    "Version": "2012-10-17"
}
```

Output 1

```
$ vendor/bin/bref inito

  What kind of lambda do you want to create? (You will be
  able to add more functions later by editing
  `template.yaml`) [PHP function]:
   [0] PHP function
   [1] HTTP application
   [2] Console application
  > 0

  Creating template.yaml
  Creating index.php


[OK] Project initialized and ready to test or deploy.
```

Listing 2

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: ''

Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: 'my-function'
      Description: ''
      CodeUri: .
      Handler: index.php
      Timeout: 10 # Timeout in seconds
      MemorySize: 1024 # Relates to pricing and CPU power
      Runtime: provided
      Layers:
        - 'arn:aws:lambda:us-east-1:209497400698:layer:php-73:1'
```

The template file consists of a small header and the `Resources` section which holds our function. Its resource name is `MyFunction` and it has a type of `AWS::Serverless::Function`. To define our function's properties, we provide a set of configuration information under the `Properties` key. The table below shows the key properties for a Lambda function.

Property name	Value	Notes
`FunctionName`	`my-function`	The name of the Lambda function. Note that this is not the same same name as the CloudFormation resource name (`MyFunction`).
`CodeUri`	`.`	The path to the source code.
`Handler`	`index.php`	The PHP file containing the lambda function the Bref runtime layer will invoke.
`Runtime`	`provided`	The language runtime that will invoke the function. For Bref, this is `provided` as we provide the runtime layer to Lambda.
`Layers`	A List of layers	By default this is Bref's PHP 7.3 runtime layer from the us-east-1 region. See this list https://runtimes.bref.sh for the correct ARNs for other regions and PHP versions.

As you can see, our template defines a single resource: a Lambda function called `my-function` that lives in `index.php` within the current directory. Let's take a look at `index.php`:

```
<?php declare(strict_types=1);

require __DIR__ . '/vendor/autoload.php';

lambda(function (array $event) {
   return 'Hello ' . ($event['name'] ?? 'world');
});
```

A Bref Lambda function is defined as a closure that's an argument to Bref's `lambda()` function. It takes an array `$event` which contains the input data to the function, and you can return whatever you want as long as it's JSON serializable. This `$event` contains information about the request which triggered your lambda function[8]. In this case, we return the string `Hello` followed by the name if it exists, otherwise `world`.

Deploying Our Function

If you used the `us-east-1` region for your configuration as recommended, then we can go ahead and deploy our function immediately.

Deployment is done using the `sam` tool; however, first, we need to create an S3 bucket to store the CloudFormation stack in.

```
$ aws s3 mb s3://helloworld-rka-brefapp
```

You can name your bucket anything you like, but it must be globally unique. I like to postfix with my initials and the reason for the bucket to ensure it's unique and that I can remember what it's for.

There are two steps to deploying our application. First, we upload the code and generate a CloudFormation stack into our S3 bucket and then we deploy the stack:

```
$ sam package --output-template-file .stack.yaml \
 --s3-bucket helloworld-rka-brefapp
$ sam deploy --template-file .stack.yaml \
 --capabilities CAPABILITY_IAM \
 --stack-name helloworld-rka-brefapp
$ rm .stack.yaml
```

The output template file (`.stack.yaml`) is an intermediate CloudFormation file and isn't needed after deploy. The stack name must be unique within the AWS region, so I name it the same as my S3 bucket.

Running Our Function

To invoke our function, we can use `bref`.

```
$ vendor/bin/bref invoke my-function
   START RequestId: 4fa0f083-c02f-4b35-a23f-1e5e35d91af5
   Version: $LATEST
   END RequestId: 4fa0f083-c02f-4b35-a23f-1e5e35d91af5
   REPORT RequestId: 4fa0f083-c02f-4b35-a23f-1e5e35d91af5

   Duration: 24.15 ms    Billed Duration: 100 ms
   Memory Size: 1024 MB  Max Memory Used: 68 MB

   "Hello world"
```

To send data to our function we can pass in a JSON object to the `--event` parameter like this:

```
$ vendor/bin/bref invoke my-function \
  --event '{"name": "Rob"}'
```

Which results in the output of `Hello Rob`.

Tidying Up

We can delete the application using:

```
aws cloudformation delete-stack \
 --stack-name helloworld-rka-brefapp
```

We also need to delete the S3 bucket with:

```
aws s3 rb s3://helloworld-rka-brefapp
```

8 *lambda function: https://phpa.me/aws-lambda-other*

Other Ways to Invoke Our Function

Invoking a function with the command line or an authenticated AWS API call is not the easiest way to execute our code; usually, you want it to respond to an event of some sort. Lambda supports many different event sources, such as a queue, database change, S3 bucket change, HTTP request or a cron-type schedule. For your function to respond to an event, you need to update the template definition with the event you want to respond to. The list of event source types can be found in the SAM documentation[9].

To schedule our function at regular intervals, we add a `Schedule` event to our function in `template.yaml` like Listing 4.

We add a new *property* called *events* and can then add as many event sources as we want. In this case, we create one event source, *MySchedule* which has a rate of one minute. The schedule property may be either a cron or a rate expression as explained in Schedule Expressions for Rules[10] in the CloudWatch documentation.

We run the `sam package` and `sam deploy` commands again to deploy the change to Lambda and now our function is executing once every minute.

To prove our function is executing once per minute, we can look in the CloudWatch logs. You can do this via the AWS console on the web or via the command line using the following command to see the Lambda function execution happening every minute.

Listing 4

```
Resources:
    MyFunction:
        Type: AWS::Serverless::Function
        Properties:
            FunctionName: 'my-function'
            # ...
            Events:
                MySchedule:
                    Type: Schedule
                    Properties:
                        Schedule: rate(1 minute)
```

```
$ sam logs --name my-function --tail
```

Don't forget to remove the *MySchedule* event and redeploy to turn it off again.

Conclusion

I've given you a taste for writing serverless functions with PHP. It is a powerful paradigm, and with Bref, we can use our PHP easily on AWS Lambda. As PHP developers, we too can benefit from this environment where our code executes in response to an event, automatically scaled as required and best of all, we only pay when our code runs. There are many situations where this paradigm can be used to add functionality to an existing application or to write a brand new application such as an API.

As serverless applications tend to utilize other services, in part two of this series, I look at how we can write a real application which integrates a Lambda function with AWS S3 cloud storage and the CloudFront CDN in order create a static website updated with new images from Flickr.

Rob Allen is a software consultant and developer with many years experience and writes code in PHP, Python, Swift and other interesting languages. He's led Slim Framework's development team and contributes to rst2pdf, Apache OpenWhisk and other open source projects. Rob is a published author and based in the UK where he runs Nineteen Feet Limited[11], focusing on API development, training and consultancy. In his spare time, Rob blogs at akrabat.com[12] and can often be seen with a camera in his hand. @akrabat

Related Reading

- *Moving a Monolith to AWS* by Keanan Koppenhaver. May 2018. https://phparch.com/article/moving-a-monolith-to-aws/
- *Community Corner: What's the Fuss About Serverless?* by James Titcumb. April 2018. https://phpa.me/april-2018-serverless

9 *SAM documentation:* https://phpa.me/sam-event-types

10 *Schedule Expressions for Rules:* https://phpa.me/cloudwatch-events-rules

11 *Nineteen Feet Limited:* http://19ft.com

12 *akrabat.com:* https://akrabat.com

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe

php[**architect**]