www.phparch.com

August 2016
VOLUME 15 - Issue 8

php[architect]

Unleashing Automation

Deploying with Ansible

Using Etsy/Phan for Static Analysis

How We Automate With Slack

ALSO INSIDE

More Than Just an "OK" Dev

What's Your Code Tolerance?

Connecting Your Charts to a MySQL Database

Community Cor
The Tangled We

Security Corner
The Hitchhiker's
Authorization

finally{}:
We are in a Tech Recession

FREE Article!

How We Automate With Slack

David Stockton

Like it or lump it, Slack has taken over as the corporate and community communication tool of choice. User groups, companies, interest groups, families, and friends use the tool to chat, share information, memes, and gifs, coordinate meetings and more. As a tool for corporate communication, Slack has become irreplaceable for many. And with its easy-to-use APIs, it's also possible to use Slack to automate common and tedious operations, as we'll see in this issue.

What's Slack?

For those of you who haven't run into Slack yet, it's a communications platform that is similar to IRC, but with some notable improvements. First of all, there's history, so you can look back at previous conversations even if you weren't connected to the server when the conversations took place. There are, of course, workarounds and programs that can archive IRC chat, but with Slack, you don't need to worry about it. This history is limited to 10,000 messages unless your Slack is a paid account, though. Slack supports direct, person-to-person communication, channels for different topics, as well as ad hoc group chats where you can enter a private chat with a small group for any purpose. You can easily search the archives, it has excellent mobile apps, and it's easy to use and understand.

In short, it's sort of a combination of Instant Messaging with chat rooms, with a splash of what email could be used for. If that's where it ended, it wouldn't be anything special. What really makes it shine, though, are the integrations. In IRC, you can add bots that listen to what people say in the channel and react with posting responses back in the channel. Your bot can also receive or react to external events and post messages into an IRC channel. Everything an IRC bot does is essentially the same as what a very attentive person logged into the channel could do. With Slack, you can incorporate bots as well, but there is also an API and other ways to enhance what you're able to do.

Slack Integrations

Currently, I'm in 12 different Slack teams. I have three related to my company, one for my user group, one for user group leaders, one with over 1,200 people who are developers in Denver, three for other companies, one for family, one for a book, and one for a specific PHP interest group. The team I spend the most time in is the main one for the company I work for and we have quite a few integrations. Integrations allow you to enhance the capabilities of the chat in various ways. We have integrations that add calculator, dictionary, thesaurus, gif postings, lunch coordination, Twitter, Pomodoro timers, IFTTT[1], and more.

For the more serious business-y integrations, we have a Bitbucket integration which posts information about pull requests—when they are posted, comments that are added, when they are merged, and more. We have an integration that can start a Google Hangout session in a channel or between two people who need to chat. From Jenkins, we get messages when jobs start and finish along with information about whether builds were successful, how many tests were run, and so on.

From JIRA we get information about new stories and tickets, and updates to ticket statuses. All this information is posted into specific channels based on its relevance. In some cases where there are a lot of pull requests or jobs, we've created a side-channel specifically for the posts from Bitbucket, JIRA, and Jenkins.

Up until now, everything I've mentioned is available with just a few clicks. Many of the integrations require only a single click and you've added the functionality to your Slack team. However, what I find more interesting are custom integrations that can help save time and make tedious tasks more enjoyable. That's what I want to discuss here.

Custom Slack Integrations

When Slack came out, I spent some time learning the API[2] and making some fun, silly and—sometimes—useful integrations. Back then, you could trigger a message to be sent to your own script in a couple of ways. You could either tell Slack to send every message in a single channel to your script, or you could send messages in any channel(s) that started with a particular trigger word to your script. Your script could then process the message and respond with messages of its own or cause something to happen.

My first few integrations mirrored several common IRC bot plugins. I built integrations to simulate dice rolls whenever someone posted something like `slackbot: roll a d6` to a channel. I built integrations to the JIRA API so that when people mentioned JIRA tickets, the bot would post the title, owner, priority, and status in the channel so you wouldn't have to search JIRA to determine what the ticket was about. I built a "karma" integration that kept score for any person or phrase that was posted preceded by "++". It has become our way of showing appreciation or respect for someone posting a particularly insightful or funny message.

Later on, Slack added a new feature known as "slash commands." These are commands you can post in any channel or chat that start with a forward slash. When you register a slash command, Slack will post a payload which includes the full message, the user who sent it, the channel it was in, and other metadata to whatever endpoint you choose.

Webhooks

For a number of years, I've wanted a server that was externally available on the internet and had access to some of our internal servers—at the very least, our Jenkins server. This is because all the services we use will send what's commonly referred to as "webhooks" when certain interesting events happen. Bitbucket will send a payload when pull requests are created, updated, or merged, or when comments are added, and more. This allowed us to make actions

1 *IFTTT:* https://ifttt.com
2 *The Slack API:* https://api.slack.com

occurring in Bitbucket and JIRA to cause Jenkins builds to happen immediately. Until we had this server, the best I could do was set up polling, so that twenty-four hours a day, seven days a week, every minute or two our Jenkins server would ask Bitbucket if there were any new pull requests or comments across more than a hundred repositories that the Jenkins server should know about. To me, that seems incredibly inefficient. I'd much rather have these services tell me when something happens that I care about. That's significantly more efficient, and I get to know about the relevant changes in a much more timely manner.

Soon after getting this server in place, we started setting up these webhooks for several of our repositories. We used Zend Expressive to inject the incoming webhooks and output API calls into Jenkins to start jobs. Of course, this meant creating new endpoints for each type of work we wanted to do. Since we were using a middleware approach, each little bit of middleware gets to do one job and then pass off the request to the next piece, and so on. Once the middleware we needed was built, it could be reused to more quickly set up a route and configure a middleware stack for the next webhooks integration.

But still, it was not what I dreamt of, which was something like an IFTTT interface where I could drag and drop various events and filters together with the consequences of those events at will. I wanted to be able to build any sort of integration I might need, as long as I had the building blocks of the middleware required to perform each task. We're not to that point yet, but we're getting there, and it's going to be amazing. Also, I want to be clear, while many of the ideas and architecture described in this article came from my brain, and an insignificant bit of the code came from me, the vast majority of the implementation of the services I'm going to talk about were written by my brother, Dann. He's done an amazing job getting this platform to where it currently is, and in a very short amount of time. It would not be what it is without him. So when I say "we," the credit really belongs to Dann.

Our First Slash Command

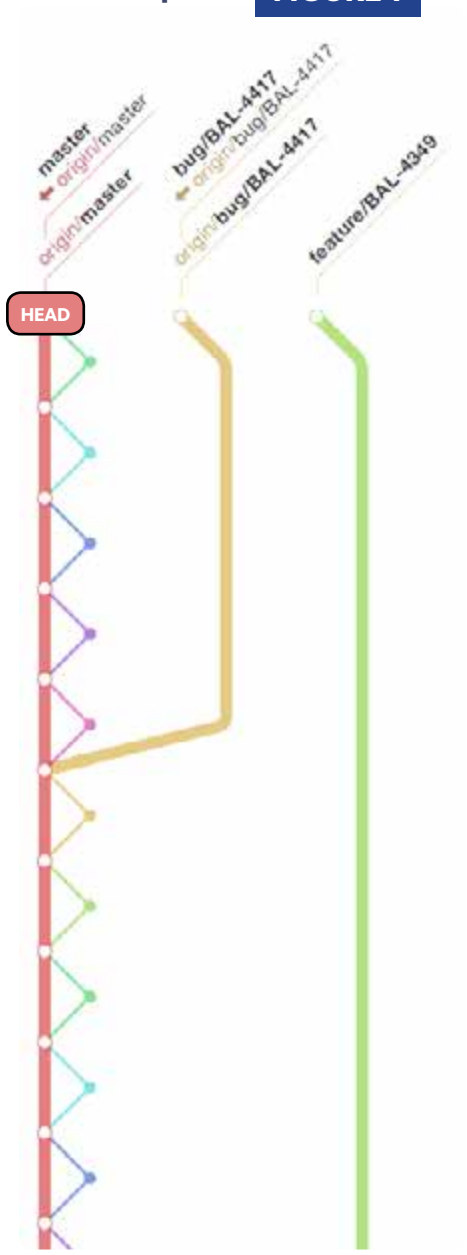
One day a few months ago, we had a team lunch. One of our remote developers needed a feature branch created by QA. Now, this team was not using git and their workflow required the feature branch to be created by QA, since developers did not have write access to the repo. I watched as our QA struggled with Bitbucket's interface on his phone, first trying to log in, then navigating to the correct repo, and finally creating a branch. It was at that lunch we decided to build a Slack slash command that would create a branch. As long as they had the Slack client, creating a branch could be as simple as writing `/branch <branch name> <repo>`. That command was finished later that day. Now QA was able to create branches easily from anywhere. We could even allow developers to create their own branches, even though they technically were not allowed to write code to this "golden" repository.

What's Next

Our other teams recently transitioned from Mercurial to git, and along with the version control changes, there were some workflow changes we could integrate because git made them possible. For the most part, the teams using git follow a rebase and squash workflow. When the code is ready to be merged, it should be a single commit branch off of the tip of the target branch. This means if several developers are working at the same time, they could each have pull requests branched off the same point in the source repository. When either of these is merged, the other branch is no longer on the tip of the target branch. We like to make each branch "zero behind" and "one ahead." Merging in this way means our source control history has a very clean "saw-tooth" style with each "tooth" being a single feature or bug fix, and the valleys between the teeth being the merges.

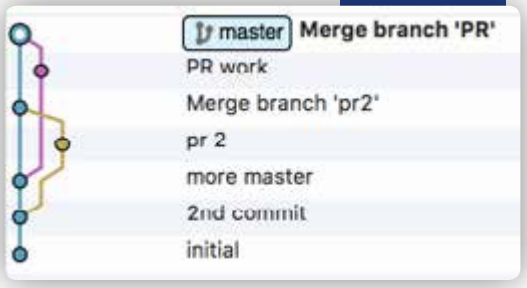
On all of our teams, the QA group was in charge of merging pull requests so they could control their workload and ensure that they've tested what they need to before deployment. In order to keep this saw-tooth pattern, it was necessary for QA to check the pull requests and ensure that they had the right number of approvals and nothing indicating that the code should not be merged. Then they had to change from the pull request screen to the branches screen and make sure the pull request was zero behind and one ahead. In other words, there was a lot of tedious checking needed to make sure that code was merged in the "sawtooth" and not the "foxtrot" pattern (shown in Figure1). It was an error prone process, as well, so we decided to automate it. We added onto our webhooks platform to do automatic merges when the right people have approved, the build is passing, and the branch is "0 1". Now, whenever a pull request is approved or updated, or a comment is deleted, the computer will check these criteria and merge automatically.

Saw-tooth pattern **FIGURE 1**



Sometimes, a pull request has all the approvals it needs and the builds are passing, but it's behind because something else merged in front of it—or it wasn't rebased

Foxtrot pattern **FIGURE 2**



to start when the pull request was created. Using the Slack APIs, we could notify the original developer that they needed to rebase their code. We decided to go one step further. So we built a Slack `/rebase` command to allow a Slack chat message to cause a rebase. We sent this command to the developer along with some information about what was happening. This allowed the developer to copy/paste the command back into Slack and our server would rebase on the developer's behalf, triggering a series of builds and checks, and ultimately merging the code if all the checks worked out.

Recently, Slack added a new bit of functionality allowing integrations to send messages with buttons. These buttons can do whatever you want them to. So, as of this afternoon, instead of having to copy/paste a slash command, developers can simply click a button and the server will rebase on their behalf. They don't need to stop and stash to rebase or context switch at all. Clicking the button makes it extremely simple, and making it simple means it gets done.

More Integrations

Additionally, we've built integrations allowing us to start Jenkins jobs from a Slack slash command. Since our deployments are controlled by a Jenkins job, it means certain people can deploy code from anywhere they have the internet without the need to connect to a VPN. We've built a JIRA slash command that does story lookups, but now through middleware and good code instead of the horrible, no good, very bad code of my original integration with JIRA. As more of the application was built, we decided that many of the new integrations were simply configuration. Automatically merging a repo is simply a matter of knowing what approval rules you want, potentially along with the source and destination branches. The middleware stack is the same as every other automatic merge. The same sort of thing goes for each Jenkins job that is kicked off automatically due to an event like a pull request creation, a comment added to the effect of "test this please" or "WAI U NO WORK?!?", or a merge happening. This led to the creation of two more slash commands that are, in effect, slash commands to create new integrations. We now have an `/auto_merge` and an `/auto_jenkins` command that build configuration for our middleware stacks to automatically merge or automatically kick off Jenkins jobs. These can now be done completely in Slack, requiring no code deploys or manual configuration updates. At this point, the application is entirely an API. There is no UI (yet) for anything. Everything is done through Slack or triggered by an application webhook.

This means that under normal circumstances, a developer can write their code, push up a pull request, and move on to their next task. They will be notified if anything needs to be done later, such as rebasing, or fixing things if tests failed, but they don't need to remember to go back and look at the code they completed. If they need to rebase, they can click a button. No one is held up waiting for it; there's essentially no context switching to rebase code, even if the developer is working on a different feature. Additionally, Jenkins jobs and deploys can be kicked off with a simple message from anywhere we have access to Slack.

Queuing

Fairly early on, we found that we could not have our application do the work as a direct response to the incoming request. The reason for this is that typically, webhooks require a quick, or relatively quick, response or they assume an error happened. For Slack, your application must respond in under 3 seconds. For Bitbucket, it has 10 seconds. The reason for this is that those services are sending out thousands of requests to their customers at any given time. They don't want to have their servers bogged down while an endpoint doesn't respond.

Since several operations may take a while, we found Queuing and "offline" asynchronous processing were necessary in order to respond quickly. This means for any given request, we do some minimal validation to ensure the request is legitimate and contains the information necessary to perform the action; then the request is shipped off as a message to a RabbitMQ server. On the other end of the queue is a long-running PHP job which picks up the request and makes a new HTTP request to the server again, but with a minor modification allowing it to bypass the queuing middleware without actually queuing the job. With incoming Slack requests, there's an endpoint we receive allowing us to send messages related to a slash command for up to 30 minutes after the initial slash command was issued. Our response back to Jenkins, JIRA, and Bitbucket is essentially responding, "Yup, that looks like it's a legit message," rather than a response indicating that all the work was done.

And Even More Better

Since Slack has an API and other services have APIs that provide useful functionality, the possibilities for integrations between Slack and other services, webhooks and Slack, or any other which way are virtually endless. We've built slash commands allowing us to explore Twilio logs, and look up information via our internal applications' own APIs. We also have a command which I think is one of the coolest, even if it's not necessarily the most practical.

We have a `/burrito` command. I'll say it again because it's awesome. We have a `/burrito` command in Slack. This command will literally cause an actual real-world burrito to be created and prepared for pick up. You can configure the burrito with your favorite ingredients and place the order without leaving the Slack interface. You can even re-order things you've already ordered in the past to save time. As far as silly level, this one ranks up there, but it works, it's awesome, and it's delicious.

A Brief Bit of Middleware

If you're not familiar with middleware, here's a super brief intro. A web request comes in, and it is transformed or acted upon little by little, through subsequent pieces of middleware, until a response is returned. That response passes back through each of the pieces of middleware it went through on the way in, until the response is sent back to the caller. A completely generic middleware could look like this pseudocode:

Of course, it's not necessary to build middleware that requires both a request and a response, but in a general sense, middleware will receive a request and ultimately return a response. Middleware is going to potentially modify the request or do something because of it, then pass it on to the

LISTING 1

```
01. function PseudoMiddleware(Request $request,
02.                           Response $response, $next) {
03.     // do something with or modify request
04.
05.     // send in to next middleware
06.     $response = $next($request, $response);
07.
08.     // modify the response on the way back out
09.
10.     // return it
11.     return $response;
12. }
```

next middleware piece. The response that's returned can be modified, or returned unchanged from the inner middleware. Because each middleware is so simple, the names will likely be helpful in understanding how some of our integrations work even without seeing the internal code. Some complication arises when you start dealing with errors and exceptions, but mostly that's not important right now. Here's an example of one of the middleware stacks we use. The extra levels of indentation indicate that a particular middleware is made up of other middleware. Here's our stack for automatically merging a pull request based on approvals and what-not.

```
* ValidatorMiddleware
  * ValidateBody
  * ValidateKey
  * ValidateEvent
  * ValidateRepo
  * ValidateDestination
* QueueRequestMiddleware
* BitbucketMiddleware
  * BitbucketApprovalCheck
  * BitbucketSpecificUserApproval
  * BitbucketCommentCheck
  * GitFetchCommand
  * GitVerifyBranch
  * GitAheadBehindCommand
  * BitbucketBuildStatusCheck
  * BitbucketMerge
  * JenkinsBuildWithParameters
```

As you can likely surmise, the `ValidatorMiddleware` is all about ensuring whatever comes in matches what we expect. The `ValidateKey` middleware ensures that the request has a secret shared key that the sender (Bitbucket) sends over. If any of these validators doesn't pass, the stack will return early and no further work will be done. The `QueueRequestMiddleware` puts the request into the Rabbit-MQ queue. On the other side, the whole process starts from the top, meaning the validation will run again, but it's quick. After that, there's a number of middlewares which ensure more specific data requirements are in place. The `BitbucketApprovalCheck` ensures that a pull request has the required number of approvals. The `BitbucketSpecificUserApproval` middleware ensures any specifically named people who must approve have done so. Next is a check to ensure that there aren't any comments that prevent merging on the actual pull request, such as "wait to merge," "this PR is not ready," or even "NO NO NO NO NO." If everything has passed so far, we get to the middleware that starts the merge. It runs a fetch, verifies that the branch we want to merge exists, and then checks that the branch is zero commits behind and one ahead (or if this was disabled, it passes through untouched). It checks to make sure all the Jenkins builds associated with this pull request have passed. It will then actually perform the merge and then optionally start another Jenkins build.

Each of these middleware pieces can be used to create other stacks for different purposes. It's a powerful way to build small bits of single-purpose code that can be used in a number of different ways. To me, it completely makes sense for this purpose. I'd recommend looking into middleware for building applications if you're not already familiar with it. It makes things seem a lot simpler, especially if you're used to working in a full stack framework.

LISTING 2

```
<?php

namespace App\Action\Bitbucket;

use App\Utility\Error;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

class BitbucketBuildCommentCheck
{
    /**
     * @var array
     */
    private $comments;

    public function __construct(array $comments) {
        $this->comments = $comments;
    }

    public function __invoke(ServerRequestInterface $request,
                             ResponseInterface $response,
                             callable $next) {
        $serverParams = $request->getServerParams();
        if (($serverParams['HTTP_X_EVENT_KEY'] ?? null)
            == "pullrequest:comment_created") {
            $body = $request->getParsedBody();
            $comment = strtolower(
                trim($body['comment']['content']['raw'])
            );
            if (!in_array($comment, $this->comments)) {
                $error = new Error(
                    "Not building PR just because you commented.",
                    ['log' => false]);
            }
        }

        return $next($request, $response, $error ?? null);
    }
}
```

Triggering a Build

This `BitbucketBuildCommentCheck` middleware in Listing 2 allows our pull requests to have special comments which start a Jenkins. Our typical configuration for these comments are "test this please", "test this plox", "test please", "test plox", "test now","u test now", or "wai u no work". If someone leaves a comment containing one of those phrases on a Pull Request, Bitbucket sends a webhook event to an endpoint specifically intended to start a Jenkins build. If the comment is not one of those, then the middleware pipeline is stopped because the comment is normal.

Queuing a Request

Listing 3 is the `QueueRequestService` class—a service which is called from middleware. It execute after a set of minimal validations have run. Those checks include things like ensuring the body of the request is JSON and contains certain key fields. If the request passes, this middleware will serialize the incoming request into Rabbit MQ and respond back with a "Processing…" message. This message will be returned back to Slack which will show the user who made that request a message. Plus since it will be fast to do this, we can get the message back to Slack quickly before it times out. Then a worker can process the job and do the real work and send additional messages to Slack once the work is completed.

LISTING 3

```php
<?php

namespace App\Service\Rabbit;

use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\JsonResponse;

class QueueRequestService
{
    /**
     * @var AMQPStreamConnection
     */
    private $connection;

    public function __construct(
        AMQPStreamConnection $connection
    ) {
        $this->connection = $connection;
    }

    /**
     * @return JsonResponse
     */
    public function queueRequest(
        ServerRequestInterface $request
    ) {
        $channel = $this->connection->channel();
        $channel->queue_declare('process_requests', false,
                                true, false, false);

        $data = json_encode(['request' => serialize($request)]);
        $msg  = new AMQPMessage(
            $data, ['delivery_mode'
                    => AMQPMessage::DELIVERY_MODE_PERSISTENT]);

        $channel->basic_publish($msg, '', 'process_requests');

        $channel->close();
        $this->connection->close();

        // send response to client
        return new JsonResponse("Processing...");
    }
}
```

Validating Keys

`ValidateKey` in Listing 4 is another middleware component used in many routes. It is provided with a set of key/value pairs from a configuration file. When the middleware pipe passes through, this class pulls a "key" value from the request's query parameters and compares the expected key with one from the provided config based on the URI from the request. This key is a "shared secret" which we use to ensure that even if the endpoint is guessed, the key must be included or the middleware pipeline will stop here.

Conclusion: Stop Re-Doing It

If there's something you have to do regularly that is painful, tedious, boring, repetitive, or error-prone, I'd highly recommend automating it. Let the computer take care of the repetitive, boring parts. If you can integrate that automation in Slack, it provides a nice way for anyone in your Slack team to take advantage of the new commands and functionality. Stop doing everything manually, and get back cycles you can use on things the computer doesn't do well, like actually writing code. Have a great month—see you next time.

David Stockton is a husband, father and Software Engineer and builds software in Colorado, leading a few teams of software developers. He's a conference speaker and an active proponent of TDD, APIs and elegant PHP. He's on twitter as @dstockto, YouTube at http://youtube.com/dstockto, and can be reached by email at levelingup@davidstockton.com.

LISTING 4

```php
<?php

namespace App\Action\Deploy\Validators;

use App\Utility\Error;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

class ValidateKey
{
    private $keys;

    public function __construct(array $keys) {
        $this->keys = $keys;
    }

    public function __invoke(ServerRequestInterface $request,
                             ResponseInterface $response,
                             callable $next) {
        $params = $request->getQueryParams();
        $path = $request->getUri()->getPath();
        $keys = $this->keys[$path] ?? [];
        if (!in_array(($params['key'] ?? null), $keys)) {
            $error = new Error("Invalid request key.");
        }

        return $next($request, $response, $error ?? null);
    }
}
```

Want more articles like this one?

Keep your skills current and stay on top of the latest PHP news and best practices by reading each new issue of php[architect], jam-packed with articles.

Learn more every month about frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[architect]

magazine

books

conferences

training

www.phparch.com

Get the complete issue for only $6!

We also offer digital and print+digital subscriptions starting at $49/year.