Delivering deployments

In this article

Using the Deployments REST API, you can build custom tooling that interacts with your server and a third-party app.

You can use the REST API to deploy your projects hosted on GitHub Enterprise Cloud on a server that you own. For more information about the endpoints to manage deployments and statuses, see "Deployments." You can also use the REST API to coordinate your deployments the moment your code lands on the default branch. For more information, see "Building a CI server."

This guide will use the REST API to demonstrate a setup that you can use. In our scenario, we will:

- Merge a pull request.
- When the CI is finished, we'll set the pull request's status accordingly.
- When the pull request is merged, we'll run our deployment to our server.

Our CI system and host server will be figments of our imagination. They could be Heroku, Amazon, or something else entirely. The crux of this guide will be setting up and configuring the server managing the communication.

If you haven't already, be sure to download `ngrok`, and learn how to use it. We find it to be a very useful tool for exposing local applications to the internet.

> **Note:** Alternatively, you can use webhook forwarding to set up your local environment to receive webhooks. For more information, see "Using the GitHub CLI to forward webhooks for testing."

Note: you can download the complete source code for this project from the platform-samples repo.

Writing your server

We'll write a quick Sinatra app to prove that our local connections are working. Let's start with this:

```
require 'sinatra'
require 'json'

post '/event_handler' do
  payload = JSON.parse(params[:payload])
  "Well, it worked!"
end
```

(If you're unfamiliar with how Sinatra works, we recommend reading the Sinatra guide.)

Start this server up. By default, Sinatra starts on port `4567`, so you'll want to configure `ngrok` to start listening for that, too.

In order for this server to work, we'll need to set a repository up with a webhook. The webhook should be configured to fire whenever a pull request is created, or merged.

Go ahead and create a repository you're comfortable playing around in. Might we suggest @octocat's Spoon/Knife repository?

After that, you'll create a new webhook in your repository, feeding it the URL that `ngrok` gave you, and choosing `application/x-www-form-urlencoded` as the content type.

Click **Update webhook**. You should see a body response of `Well, it worked!`. Great! Click on **Let me select individual events.**, and select the following:

- Deployment
- Deployment status
- Pull Request

These are the events GitHub Enterprise Cloud will send to our server whenever the relevant action occurs. We'll configure our server to *just* handle when pull requests are merged right now:

```
post '/event_handler' do
  @payload = JSON.parse(params[:payload])

  case request.env['HTTP_X_GITHUB_EVENT']
  when "pull_request"
    if @payload["action"] == "closed" && @payload["pull_request"]["merged"]
      puts "A pull request was merged! A deployment should start now..."
    end
  end
end
```

What's going on? Every event that GitHub Enterprise Cloud sends out attached a `X-GitHub-Event` HTTP header. We'll only care about the PR events for now. When a pull request is merged (its state is `closed`, and `merged` is `true`), we'll kick off a deployment.

To test out this proof-of-concept, make some changes in a branch in your test repository, open a pull request, and merge it. Your server should respond accordingly!

Working with deployments

With our server in place, the code being reviewed, and our pull request merged, we want our project to be deployed.

We'll start by modifying our event listener to process pull requests when they're merged, and start paying attention to deployments:

```
when "pull_request"
  if @payload["action"] == "closed" && @payload["pull_request"]["merged"]
    start_deployment(@payload["pull_request"])
  end
when "deployment"
  process_deployment(@payload)
when "deployment_status"
  update_deployment_status
end
```

Based on the information from the pull request, we'll start by filling out the `start_deployment` method:

```
def start_deployment(pull_request)
  user = pull_request['user']['login']
  payload = JSON.generate(:environment => 'production', :deploy_user => user)
  @client.create_deployment(pull_request['head']['repo']['full_name'],
pull_request['head']['sha'], {:payload => payload, :description => "Deploying my
sweet branch"})
end
```

Deployments can have some metadata attached to them, in the form of a `payload` and a `description`. Although these values are optional, it's helpful to use for logging and representing information.

When a new deployment is created, a completely separate event is triggered. That's why we have a new `switch` case in the event handler for `deployment`. You can use this information to be notified when a deployment has been triggered.

Deployments can take a rather long time, so we'll want to listen for various events, such as when the deployment was created, and what state it's in.

Let's simulate a deployment that does some work, and notice the effect it has on the output. First, let's complete our `process_deployment` method:

```
def process_deployment
  payload = JSON.parse(@payload['payload'])
  # you can send this information to your chat room, monitor, pager, etc.
  puts "Processing '#{@payload['description']}' for #{payload['deploy_user']} to
#{payload['environment']}"
  sleep 2 # simulate work
  @client.create_deployment_status("repos/#{@payload['repository']
['full_name']}/deployments/#{@payload['id']}", 'pending')
  sleep 2 # simulate work
  @client.create_deployment_status("repos/#{@payload['repository']
['full_name']}/deployments/#{@payload['id']}", 'success')
end
```

Finally, we'll simulate storing the status information as console output:

```
def update_deployment_status
  puts "Deployment status for #{@payload['id']} is #{@payload['state']}"
end
```

Let's break down what's going on. A new deployment is created by `start_deployment`, which triggers the `deployment` event. From there, we call `process_deployment` to simulate work that's going on. During that processing, we also make a call to `create_deployment_status`, which lets a receiver know what's going on, as we switch the status to `pending`.

After the deployment is finished, we set the status to `success`.

Conclusion

At GitHub, we've used a version of Heaven to manage our deployments for years. A common flow is essentially the same as the server we've built above:

- Wait for a response on the state of the CI checks (success or failure)
- If the required checks succeed, merge the pull request
- Heaven takes the merged code, and deploys it to staging and production servers
- In the meantime, Heaven also notifies everyone about the build, via Hubot sitting in our chat rooms

That's it! You don't need to build your own deployment setup to use this example. You

can always rely on GitHub integrations.