Building a CI server

In this article

Build your own CI system using the Status API.

You can use the REST API to tie together commits with a testing service, so that every push you make can be tested and represented in a GitHub Enterprise Server pull request. For more information about the relevant endpoints, see "Commit statuses."

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

- Run our CI suite when a Pull Request is opened (we'll set the CI status to pending).
- When the CI is finished, we'll set the Pull Request's status accordingly.

Our CI system and host server will be figments of our imagination. They could be Travis, Jenkins, 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, 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: 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:

- Status
- Pull Request

These are the events GitHub Enterprise Server will send to our server whenever the relevant action occurs. Let's update our server to *just* handle the Pull Request scenario right now:

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

  case request.env['HTTP_X_GITHUB_EVENT']
  when "pull_request"
    if @payload["action"] == "opened"
      process_pull_request(@payload["pull_request"])
    end
  end
end

helpers do
  def process_pull_request(pull_request)
    puts "It's #{pull_request['title']}"
  end
end
```

What's going on? Every event that GitHub Enterprise Server sends out attached a `X-GitHub-Event` HTTP header. We'll only care about the PR events for now. From there, we'll take the payload of information, and return the title field. In an ideal scenario, our server would be concerned with every time a pull request is updated, not just when it's opened. That would make sure that every new push passes the CI tests. But for this demo, we'll just worry about when it's opened.

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

Working with statuses

With our server in place, we're ready to start our first requirement, which is setting (and updating) CI statuses. Note that at any time you update your server, you can click **Redeliver** to send the same payload. There's no need to make a new pull request every time you make a change!

Since we're interacting with the GitHub Enterprise Server API, we'll use Octokit.rb to manage our interactions. We'll configure that client with a personal access token:

```
# !!! DO NOT EVER USE HARD-CODED VALUES IN A REAL APP !!!
# Instead, set and test environment variables, like below
ACCESS_TOKEN = ENV['MY_PERSONAL_TOKEN']

before do
  @client ||= Octokit::Client.new(:access_token => ACCESS_TOKEN)
end
```

After that, we'll just need to update the pull request on GitHub Enterprise Server to make clear that we're processing on the CI:

```
def process_pull_request(pull_request)
```

```
  puts "Processing pull request..."
  @client.create_status(pull_request['base']['repo']['full_name'],
pull_request['head']['sha'], 'pending')
end
```

We're doing three very basic things here:

- we're looking up the full name of the repository
- we're looking up the last SHA of the pull request
- we're setting the status to "pending"

That's it! From here, you can run whatever process you need to in order to execute your test suite. Maybe you're going to pass off your code to Jenkins, or call on another web service via its API, like Travis. After that, you'd be sure to update the status once more. In our example, we'll just set it to `"success"`:

```
def process_pull_request(pull_request)
  @client.create_status(pull_request['base']['repo']['full_name'],
pull_request['head']['sha'], 'pending')
  sleep 2 # do busy work...
  @client.create_status(pull_request['base']['repo']['full_name'],
pull_request['head']['sha'], 'success')
  puts "Pull request processed!"
end
```

Conclusion

At GitHub, we've used a version of Janky to manage our CI for years. The basic flow is essentially the exact same as the server we've built above. At GitHub, we:

- Fire to Jenkins when a pull request is created or updated (via Janky)
- Wait for a response on the state of the CI
- If the code is green, we merge the pull request

All of this communication is funneled back to our chat rooms. You don't need to build your own CI setup to use this example. You can always rely on GitHub integrations.