

Integrating GitHub Webhooks with Jenkins to automate unit and integration test after GitHub events for CI/CD *

1. Exposing Jenkins to the Internet

Before beginning, this step is optional if you already have a Jenkins server exposed to the internet. If so, continue to section two, otherwise keep reading. Ok, so you are working with a local Jenkins running in some local environment, outside of any cloud provider such as Amazon EC2, and you want to use this local Jenkins server to test some workflows that can be implemented in a future formal environment without having to think on closing the cloud instance to save money. But still, you need a way to expose Jenkins to outside networks since this is a requisite to work with external services such as Webhooks.

To do so, we are going to use a service called: ngrok. [Ngrok](#) can expose local servers to the internet through a public IP. The free tier can work with one online ngrok process and up to 4 tunnels. In this case, we only need one for Jenkins, so the free tier is great for our needs.

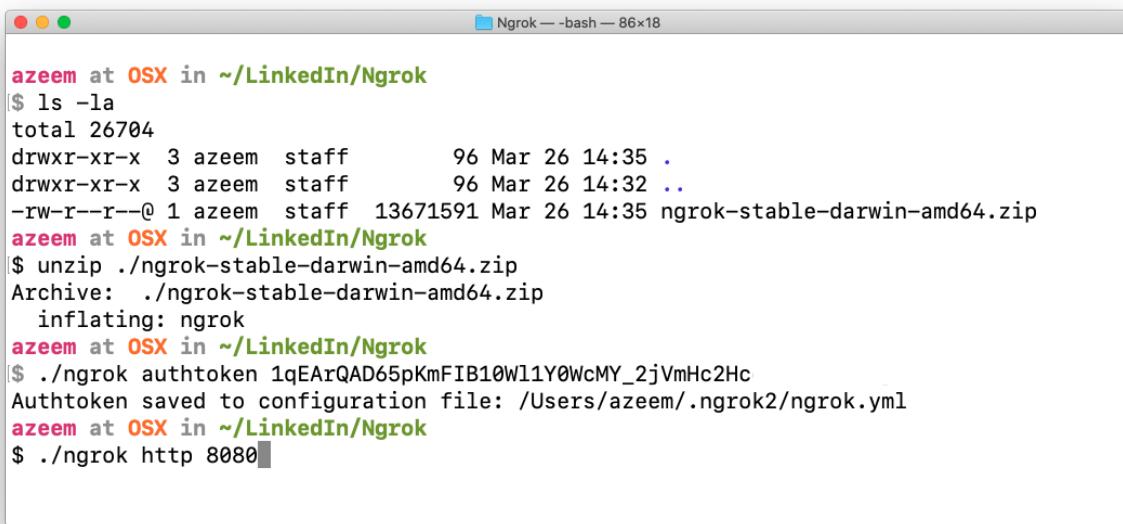
The screenshot shows the ngrok.com homepage. At the top, there's a navigation bar with links for "How it works", "Pricing", "Enterprise solutions", "Docs", "Download", "Login", and "Sign up". The main heading is "Public URLs for building webhook integrations." Below this, a sub-headline says "Spend more time programming. One command for an instant, secure URL to your localhost server through any NAT or firewall." A blue button labeled "Get started for free" is visible. On the right side, there's a session status window titled "https://katesapp.ngrok.io". It displays the command used ("./ngrok http 3000"), the session status ("online"), account information ("Kate Libby (Plan: Pro)"), and forwarding details. A blue button labeled "Ask a question" is at the bottom right of the session window.

After register, we need to download the executables

The screenshot shows the ngrok dashboard at "dashboard.ngrok.com". The left sidebar has sections for "Getting Started", "Setup & Installation" (which is selected), "Your Authtoken", "Tutorials", and "Endpoints". A callout box highlights "Setup & Installation" with the text "Get access to powerful features like:" followed by a bulleted list: "URLs that don't change", "Your own custom domains", "Access to beta features", and "and more!". A blue "Upgrade Now" button is at the bottom of this box. The main content area is titled "Download ngrok". It explains that ngrok is easy to install and provides a "Download for Mac OS" button. To the right, there are download links for Mac OS (Mac OS and Mac OS (32-Bit)), Windows (Windows and Windows (32-Bit)), Linux (Linux, Linux (32-Bit), Linux (ARM), and Linux (ARM64)), and FreeBSD (FreeBSD). Below this, a section titled "1. Unzip to install" provides instructions for Linux/Mac OS X and shows a terminal command: "\$ unzip /path/to/ngrok.zip".

Using the terminal, inside the directory we store ngrok, we continue to unzip it, add our token from the ngrok webpage, and start the server with the 8080 port corresponding to Jenkins.

Using the terminal, inside the directory we store ngrok, we continue to unzip it, add our token from the ngrok webpage, and start the server with the 8080 port corresponding to Jenkins.

A screenshot of a Mac OS X terminal window titled "Ngrok — bash — 86x18". The window shows a series of terminal commands being run. The user is in their home directory (~) and navigating to the Ngrok folder. They list all files (ls -la), download the ngrok-stable-darwin-amd64.zip file, extract it (unzip), and then run the ngrok command with their auth token. The output shows the token was saved to a configuration file.

```
azeem at OSX in ~/LinkedIn/Ngrok
$ ls -la
total 26704
drwxr-xr-x  3 azeem  staff      96 Mar 26 14:35 .
drwxr-xr-x  3 azeem  staff      96 Mar 26 14:32 ..
-rw-r--r--@ 1 azeem  staff  13671591 Mar 26 14:35 ngrok-stable-darwin-amd64.zip
azeem at OSX in ~/LinkedIn/Ngrok
$ unzip ./ngrok-stable-darwin-amd64.zip
Archive: ./ngrok-stable-darwin-amd64.zip
  inflating: ngrok
azeem at OSX in ~/LinkedIn/Ngrok
$ ./ngrok authtoken 1qEAQAD65pKmFIB10Wl1Y0WcMY_2jVmHc2Hc
Authtoken saved to configuration file: /Users/azeem/.ngrok2/ngrok.yml
azeem at OSX in ~/LinkedIn/Ngrok
$ ./ngrok http 8080
```

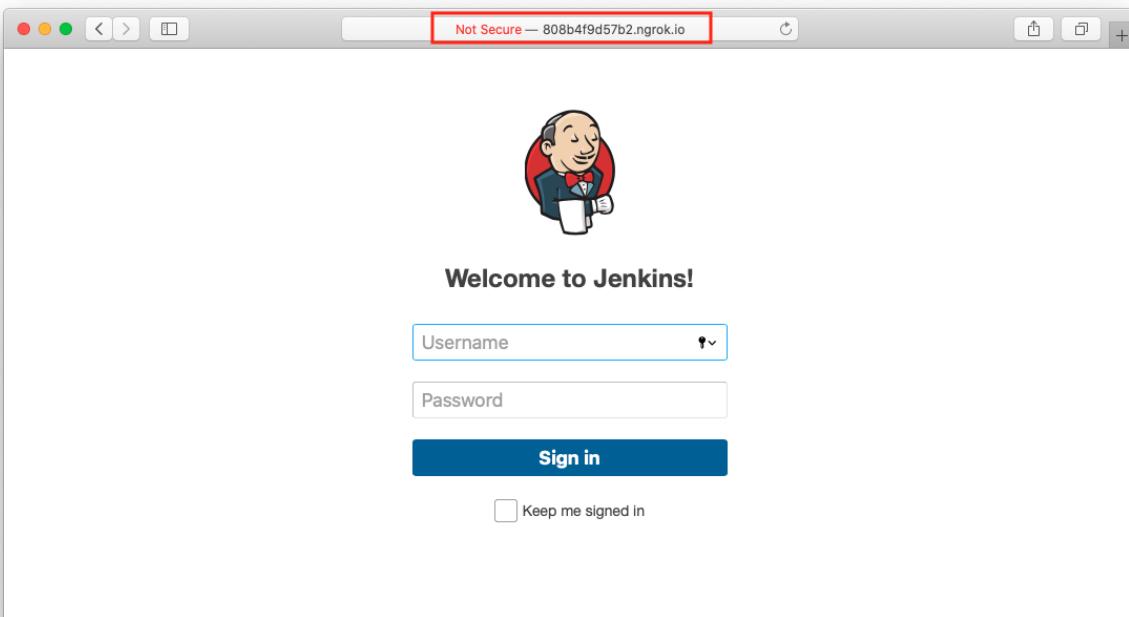
Once we press enter, the ngrok server starts and gives us a public IP, which forwards any petitions to our localhost 8080 port.

```
Ngrok — ngrok http 8080 — 86x18
ngrok by @inconshreveable                                         (Ctrl+C to quit)

Session Status          online
Account                 Oscar Azeem (Plan: Free)
Version                2.3.37
Region                 United States (us)
Web Interface          http://127.0.0.1:4040
Forwarding              http://808b4f9d57b2.ngrok.io -> http://localhost:8080
Forwarding              https://808b4f9d57b2.ngrok.io -> http://localhost:8080

Connections            ttl     opn      rt1      rt5      p50      p90
                        0       0       0.00    0.00    0.00    0.00
```

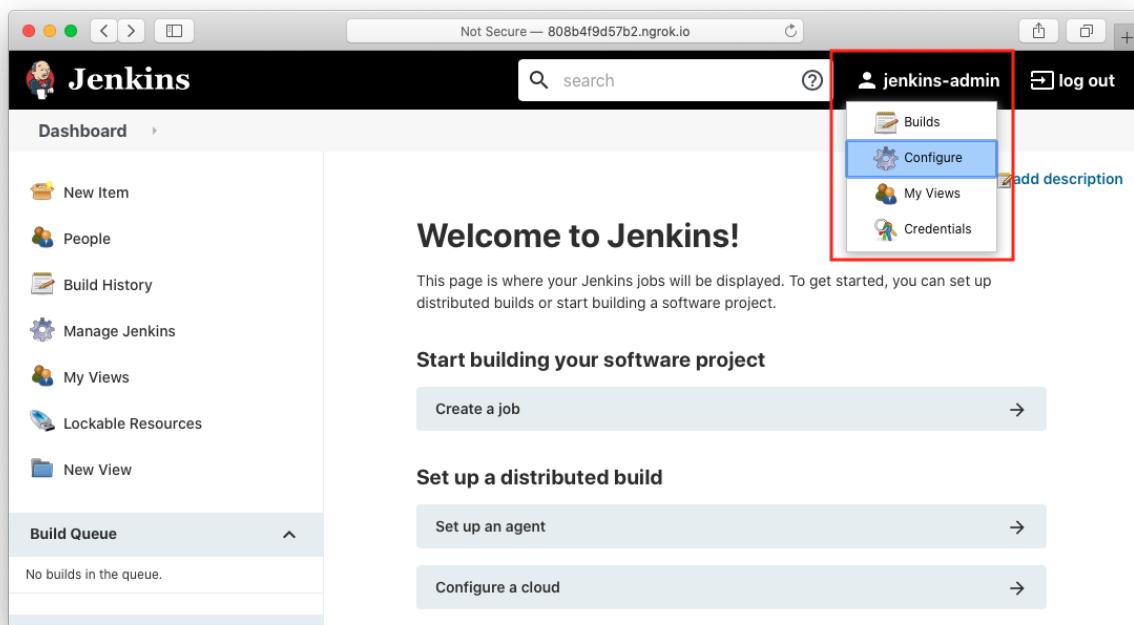
Using this direction, we can access our localhost 8080 port (Jenkins) from any outside network with our web explorer.



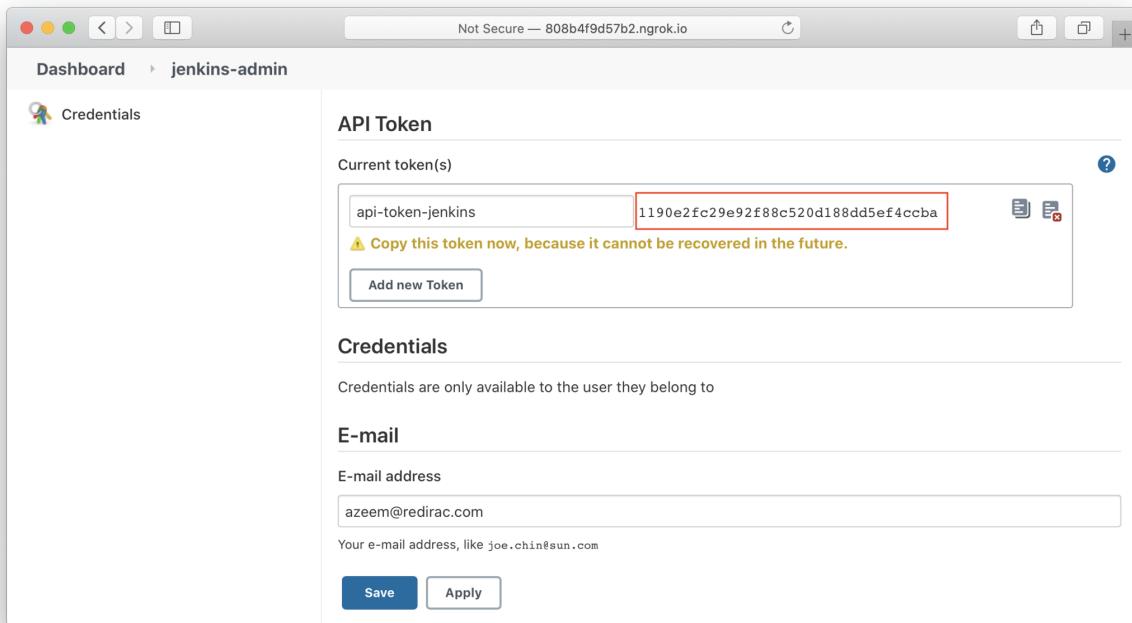
So far so good, using ngrok we have exposed our local Jenkins to the internet. The only step remaining it's to login to Jenkins and continue with the next section. 

2. Create a Jenkins API Token

To connect our Jenkins server with our Webhook, we will first need to create an API Token to authenticate from the version control provider (Github in this case). To do so, we need to click on our account and then on the configure option.



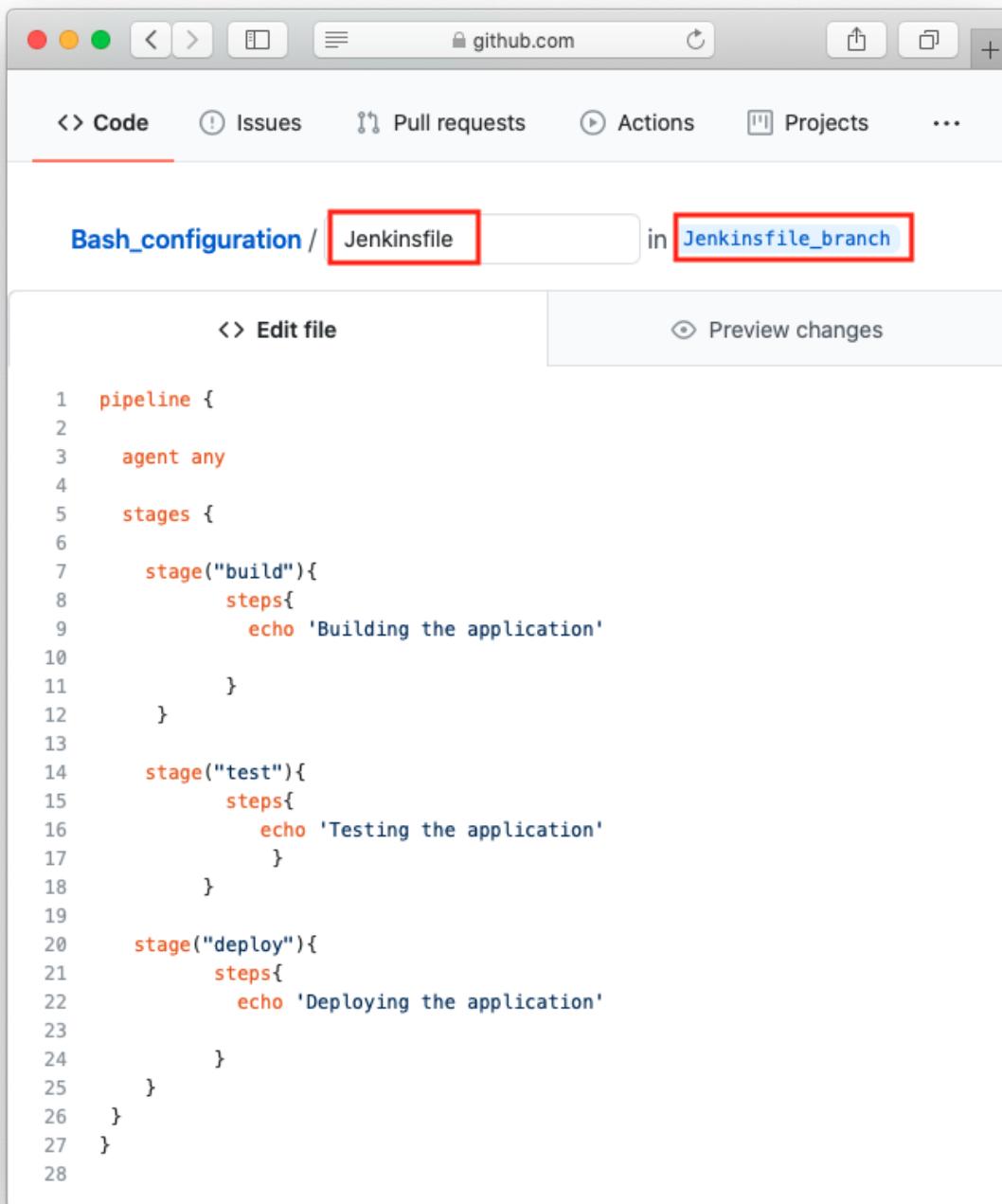
Click on Add new Token, write a name for it and then press Generate. We need to store this Token in a safe place because there is no way to recover it in the future unless creating a new one.



3. Configuring our Github repository

3.1 Adding a Jenkinsfile

Before creating a Jenkins pipeline to be triggered with the webhook, it's good practice to include a Jenkins file inside our git project so that there are already declared all the stages and steps to be validated on each run. The following is an example with three stages: build, test, and deploy if you don't already have a Jenkinsfile in your project. Each one of the three stages has a single step echoing the current step.

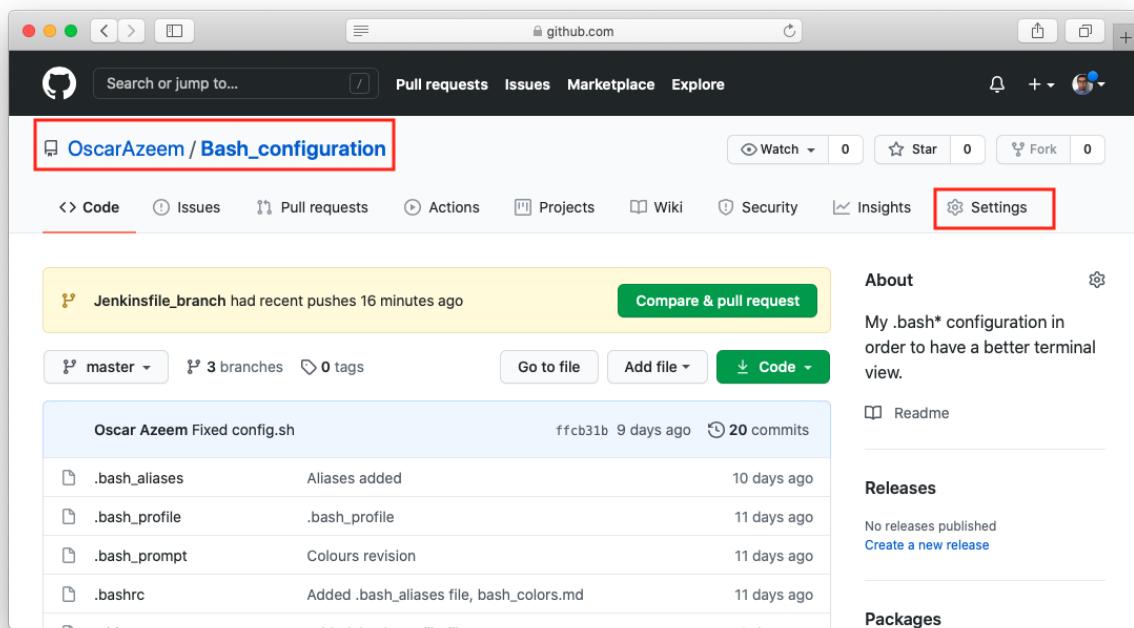


The screenshot shows a GitHub repository interface. At the top, there are navigation links: Code, Issues, Pull requests, Actions, Projects, and a more options menu. Below these, the repository path is displayed as 'Bash_configuration / Jenkinsfile' in a red box, with 'in Jenkinsfile_branch' to its right. There are two buttons: 'Edit file' and 'Preview changes'. The main area contains the Jenkinsfile code:

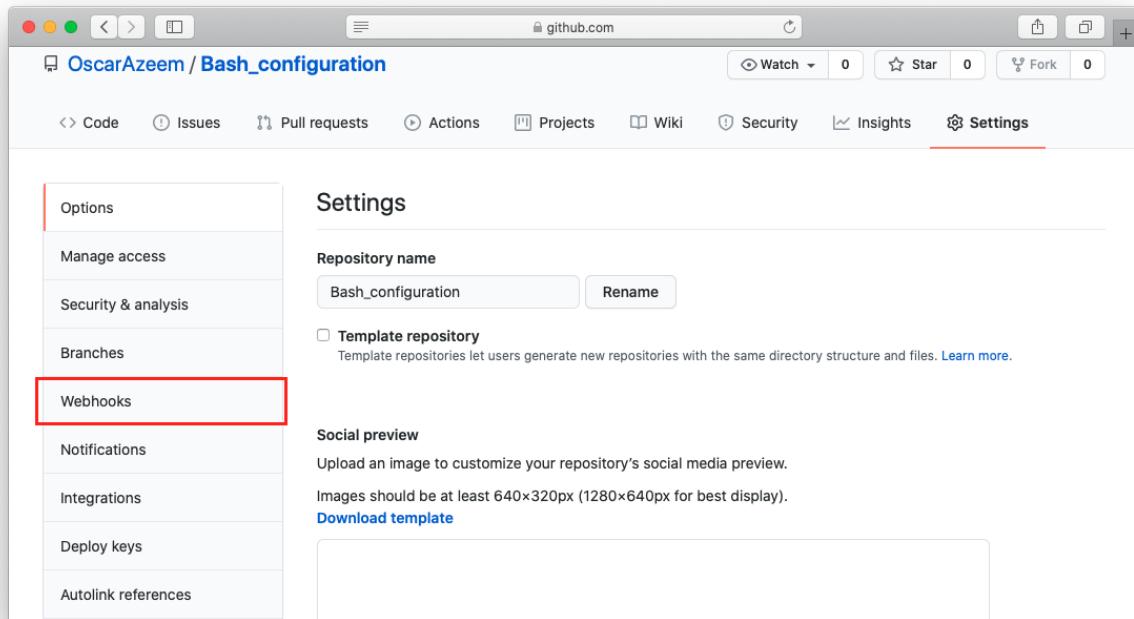
```
1 pipeline {
2     agent any
3     stages {
4         stage("build"){
5             steps{
6                 echo 'Building the application'
7             }
8         }
9         stage("test"){
10            steps{
11                echo 'Testing the application'
12            }
13        }
14        stage("deploy"){
15            steps{
16                echo 'Deploying the application'
17            }
18        }
19    }
20 }
```

3.2 Creating the Github webhook

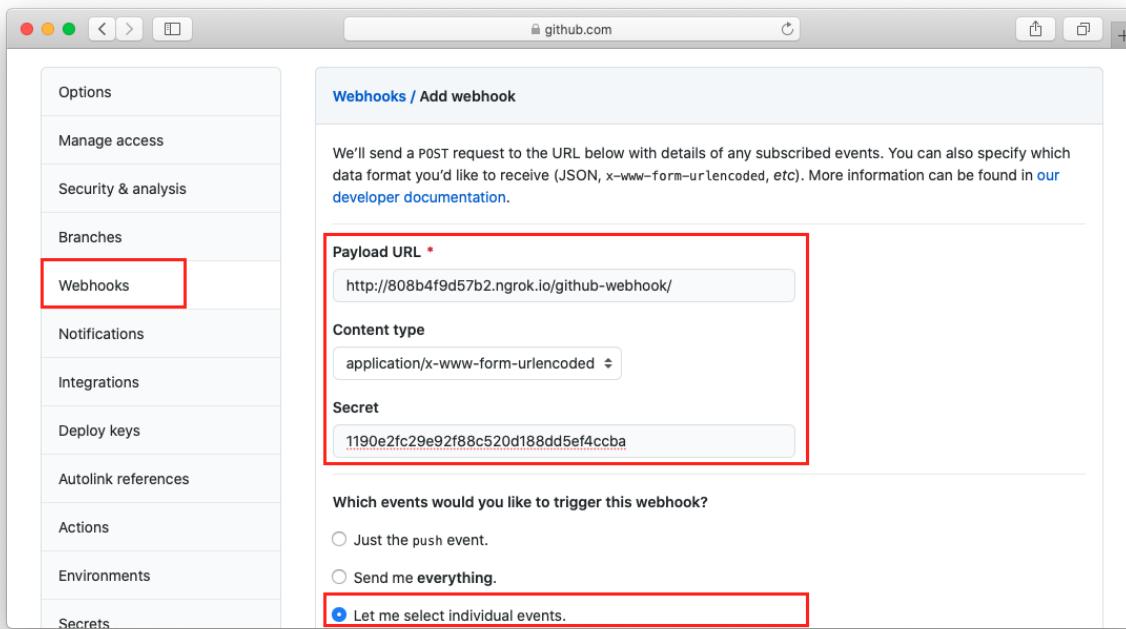
To create our GitHub Webhook, we need to open our repository and click on settings.



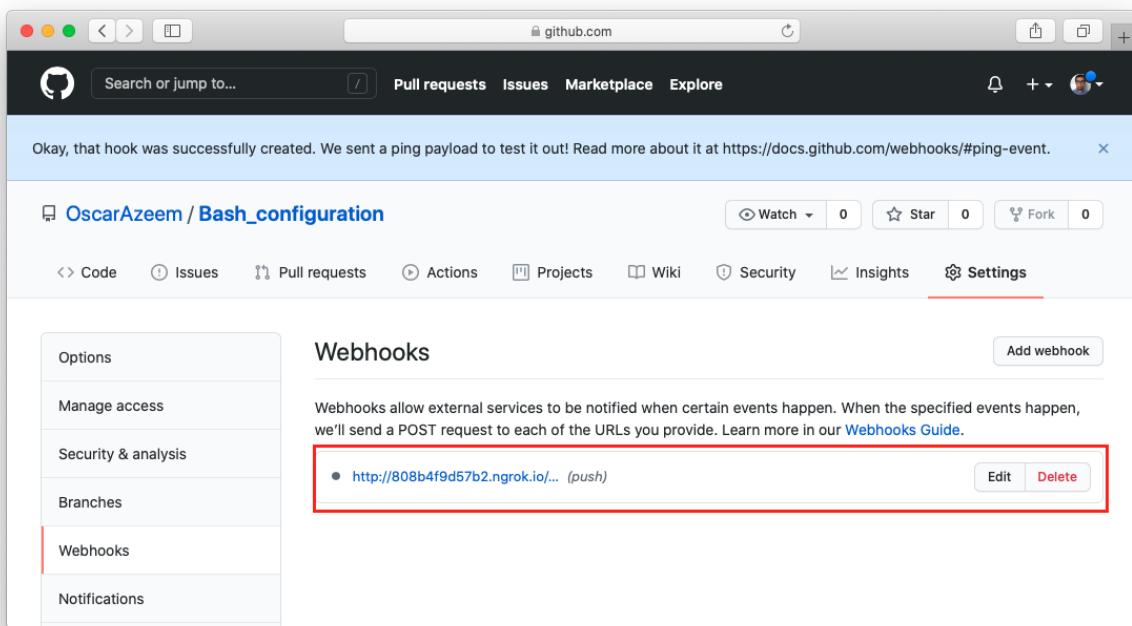
Then click on Webhooks and Add webhook



Inside the Payload URL text box, we need to write the direction to our Jenkins Server, plus: /github-webhook/. If you created a ngrok account, it'd be obtained at the end of section one. The payload must have both / to work, otherwise it'll throw some error. Inside the Secret box, we'll paste the Jenkins API Token from section two. Finally, click on: Let me select individual elements to choose the desired webhook events for our needs to trigger Jenkins.



Once created the webhook, it'll look like the following image.



3.2 Creating a GitHub Token

Since early 2021 GitHub restricts API user authentication to work only with a token or private ssh key. There is no longer possible to authenticate using our user and password, and such is the case working with Jenkins. If you are not using GitHub, continue to section 4. Otherwise, keep reading.

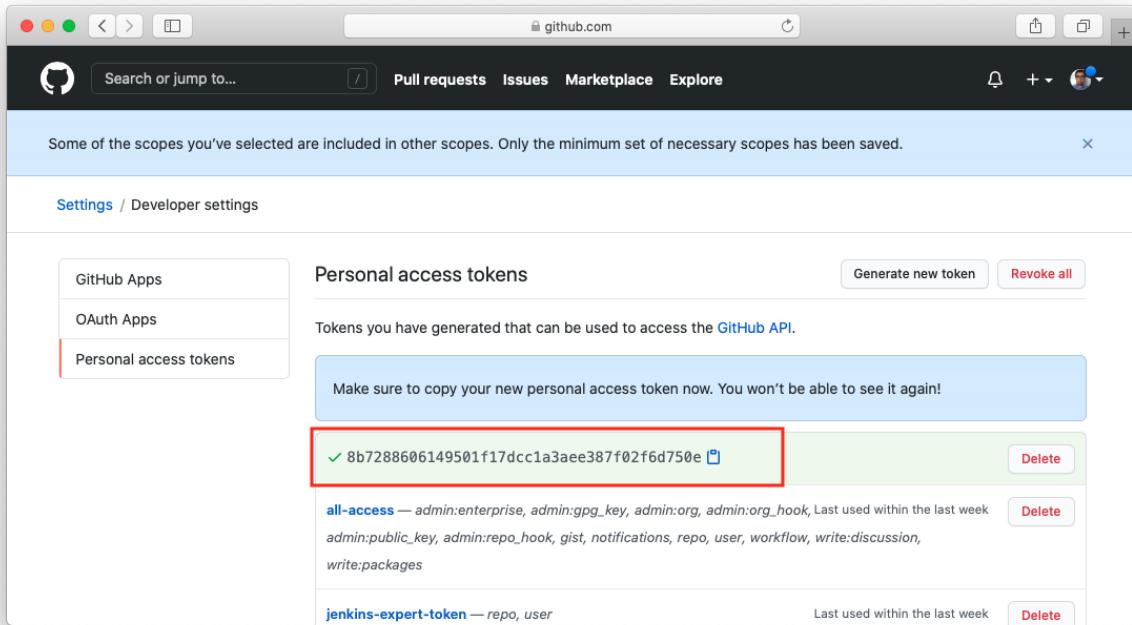
To create a token to be authenticated from Jenkins, we need to go to our GitHub profile, then click on Settings, Developer Settings, Personal access tokens (or go to this direction: <https://github.com/settings/tokens>). Once there, click on Generate new token

The screenshot shows the GitHub developer settings page. The 'Personal access tokens' section is highlighted with a red box. The 'Generate new token' button at the top right is also highlighted with a red box. The token list includes three entries: 'all-access', 'jenkins-expert-token', and 'jenkins-token-basic'. Each entry shows the scopes, last used date, and a 'Delete' button.

We give it a name plus checking the two boxes: repo and user, finally click on Generate Token.

The screenshot shows the 'New personal access token' dialog. The 'Personal access tokens' section in the sidebar is highlighted with a red box. The 'Note' field contains 'jenkins-token-repo' and is highlighted with a red box. The 'Select scopes' section shows a list of scopes with checkboxes. The 'repo' checkbox is checked and highlighted with a red box. Other checked scopes include 'repo:status', 'repo_deployment', 'public_repo', 'repo:invite', and 'security_events'. Unchecked scopes include 'workflow' and 'write:packages'.

Once the token has been generated, it'll look like the following. Again, these tokens are meant to be kept in a safe place, there is no way to recover them once lost unless creating a new one.



4. Configuring Jenkins

Let's recapitulate a little. We have our Jenkins server exposed to the internet using ngrok. Then created a Jenkins API Token so we can be authenticated with the GitHub webhook. Later we create a Jenkinsfile for our project (if there hasn't been any yet) and configure our GitHub webhook and token (for the Github Jenkins API). Now we need to add these credentials to Jenkins and create a pipeline to run the Jenkinsfile tests for our remote repository.

4.1. Adding GitHub Credentials

To add our Github credentials on Jenkins, we need to go to the dashboard, Manage Jenkins, and Configure System. If a message appears saying that our reverse proxy set up is broken, just click on dismiss. This message is due to the proxy connection using ngrok.

The screenshot shows the Jenkins Manage Jenkins interface. The left sidebar has a red box around the 'Manage Jenkins' link. The main content area has a red box around the 'Configure System' section. A message at the top says 'It appears that your reverse proxy set up is broken.' with 'More Info' and 'Dismiss' buttons. Other sections like 'Manage Plugins' and 'Manage Nodes and Clouds' are also visible.

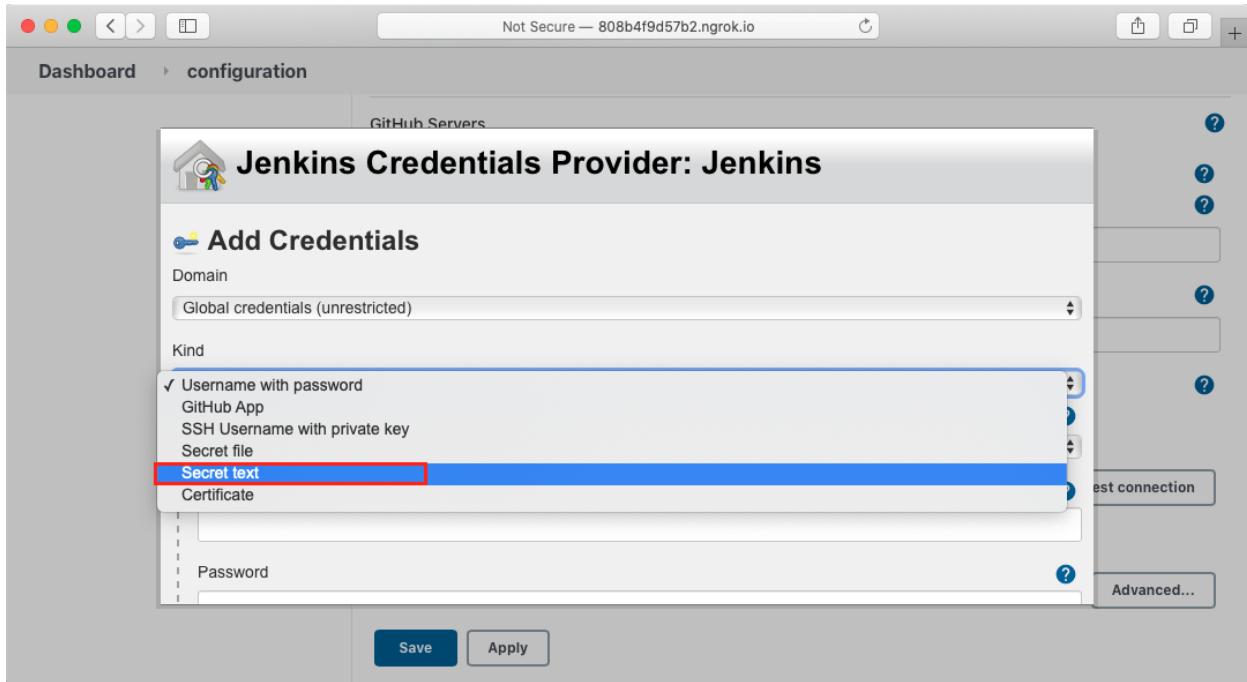
Scroll down to the GitHub section (you need first to have the GitHub plugin installed on Jenkins; if you installed all the recommended plugins at Jenkins installation, it should be there) click on Add GitHub Server, GitHub Server

The screenshot shows the Jenkins configuration interface. In the top left, it says "Dashboard > configuration". On the left, there's a sidebar with "GitHub", "GitHub API usage", "GitHub Enterprise Servers", and "Global Pipeline Libraries". Under "GitHub", there's a "GitHub Servers" section with a red box around it. Inside this box are a "Add GitHub Server" button and a "GitHub Server" button. To the right of the red box is a "Advanced..." button.

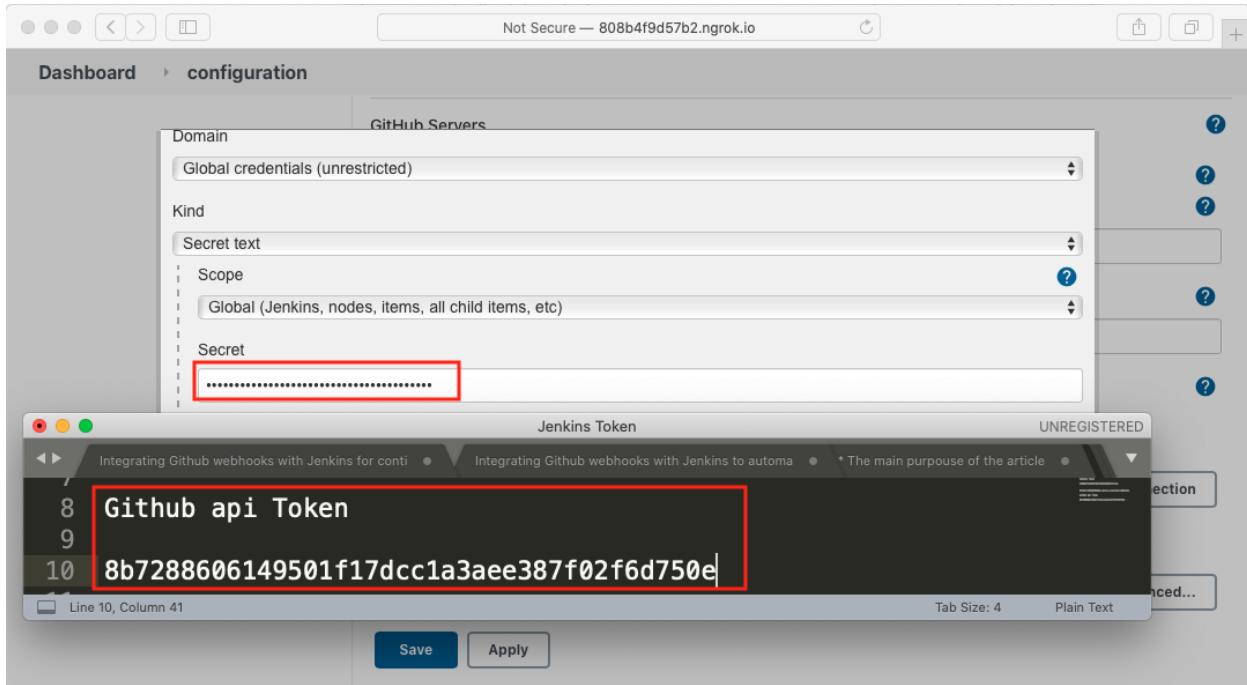
Inside the credentials section click on add, Jenkins.

This screenshot shows the "GitHub Servers" configuration for a specific "GitHub Server". The "Name" field is empty. The "API URL" field contains "https://api.github.com". Below these fields is a "Credentials" section with a red box around it. It includes a dropdown menu set to "- none -", an "Add" button, and a "Jenkins" button. To the right of the red box is a "Test connection" button. At the bottom are "Save" and "Apply" buttons.

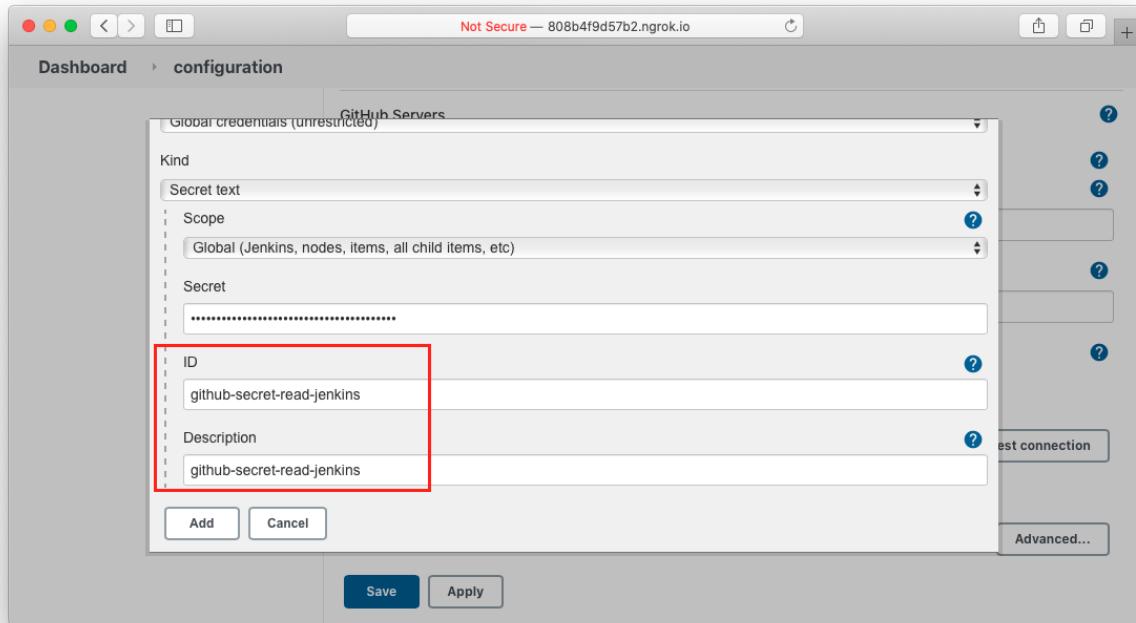
It'll appear a new window. On this window, we should select Secret text



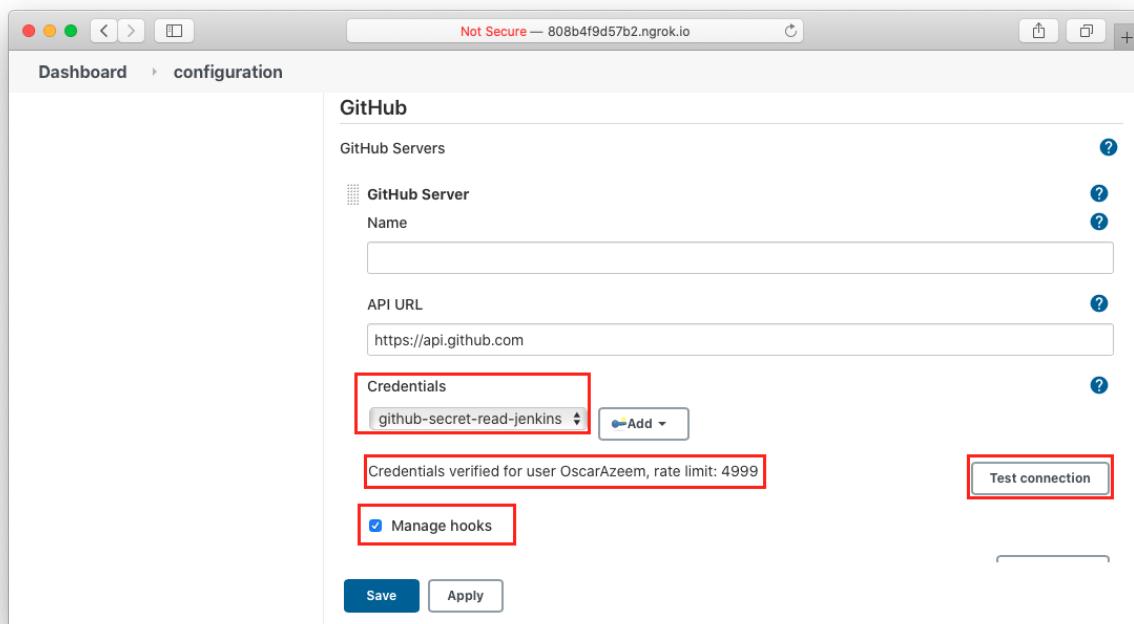
Inside the Secret box, we paste the GitHub API token obtained in section 3.2



Give it some ID and Description and click on ADD

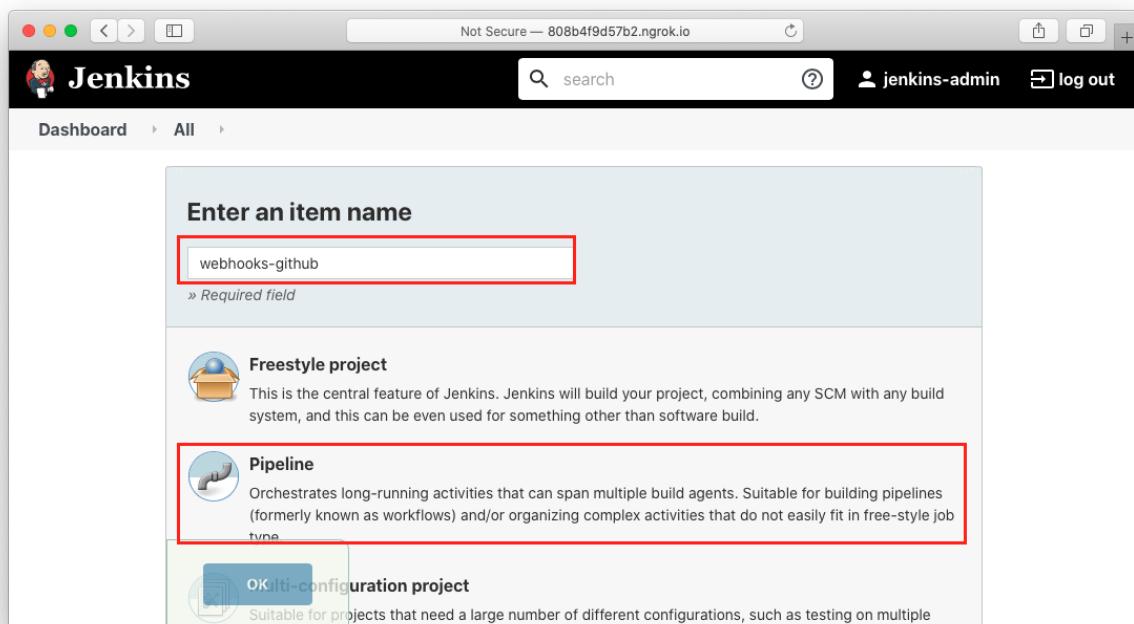


Now we can see this new credential inside the credentials option box, select it, and then click on Test connection. If everything goes right, it should say, Credentials verified for user <My-Github-User>, rate limit: <Number>. If you receive some error, recheck your GitHub API token. Finally, enable the Manage Hooks option and click on save.

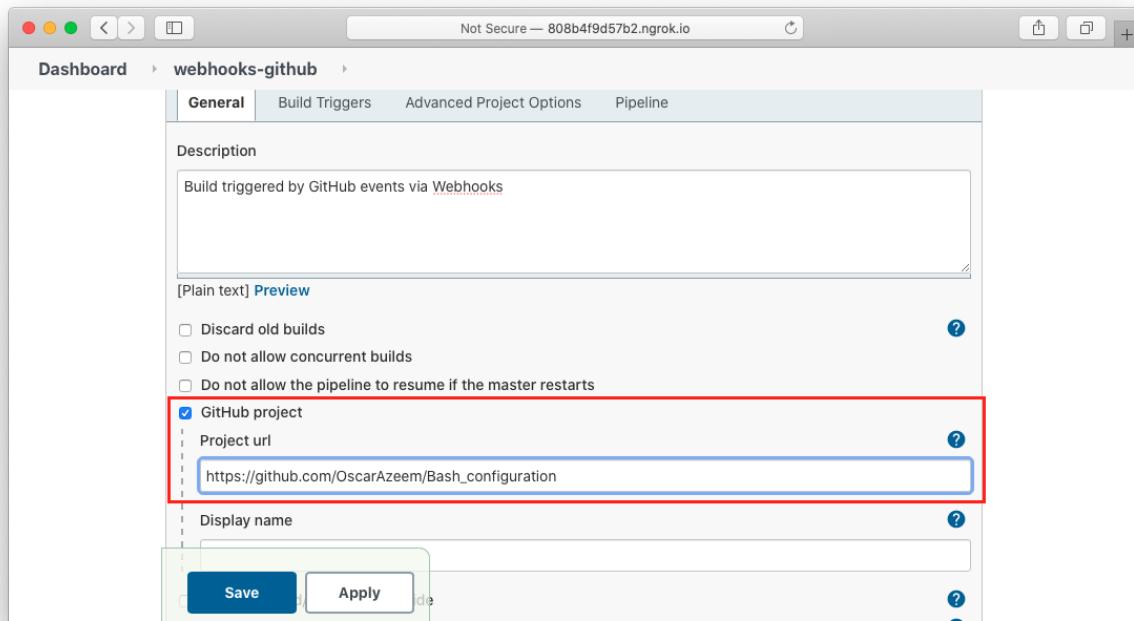


4.2 Creating a Jenkins Pipeline

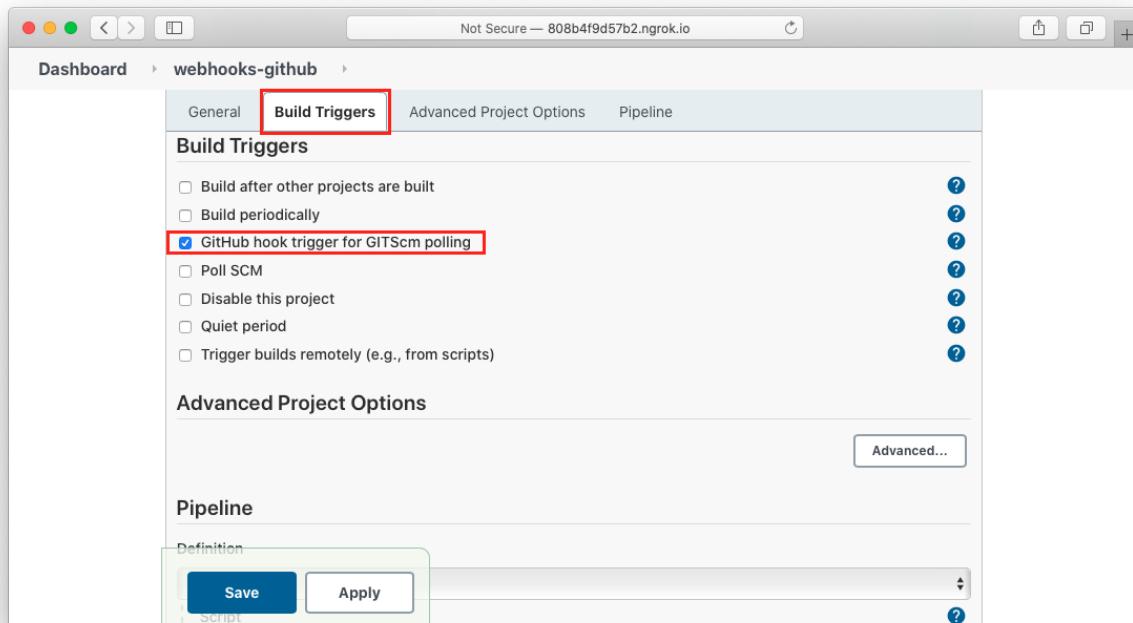
To create a Jenkins pipeline to test our webhook, we go to the dashboard, new item, give it a name and click on the Pipeline job.



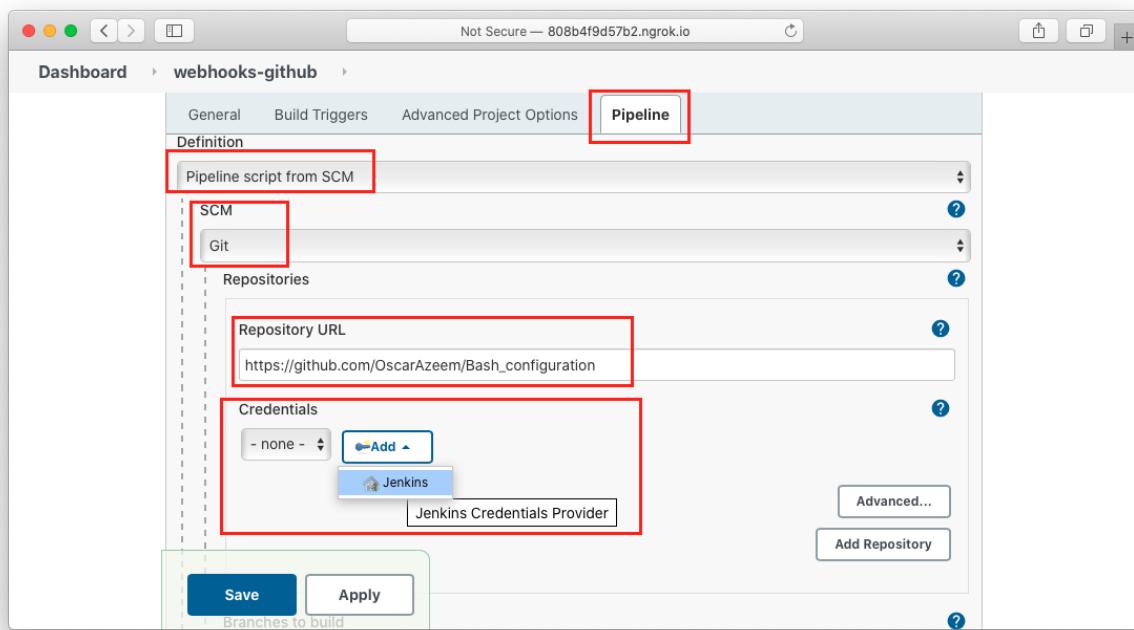
In the General Tab add a description for this pipeline, check the option: GitHub project, and inside the Project Url text box, write the URL for the remote repository to be tested.



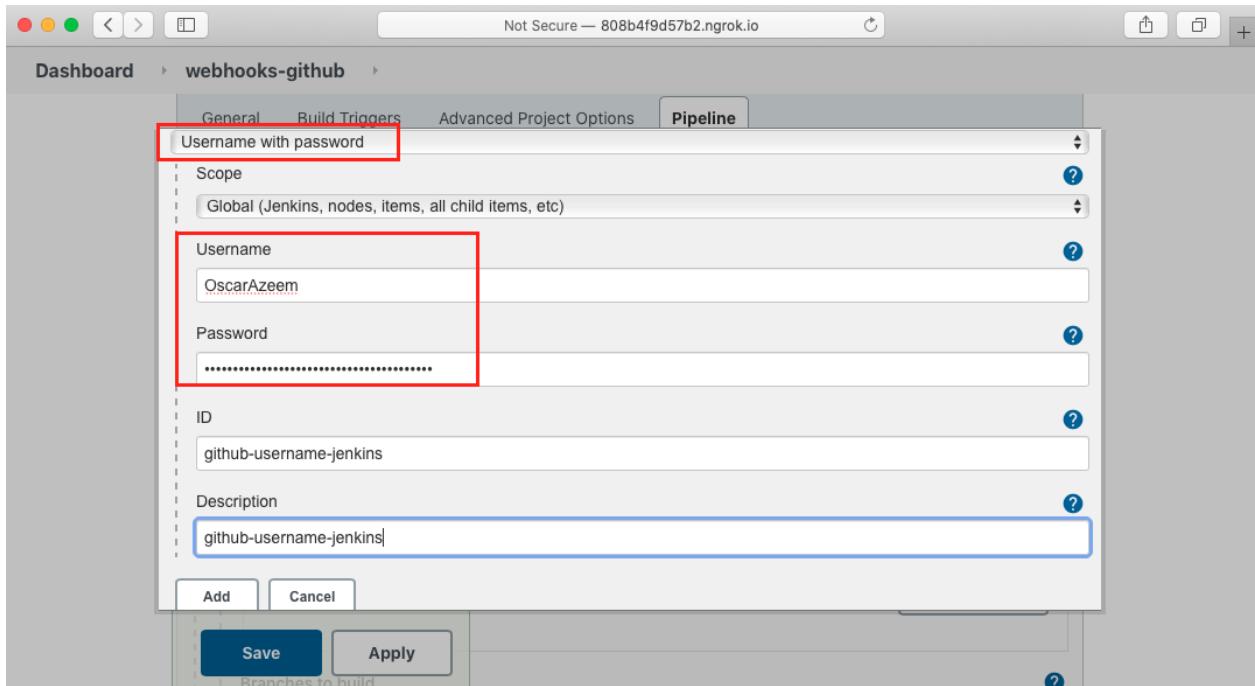
In the Build Triggers tab, check the box: GitHub hook trigger for GITScm polling. This option will enable this build to run after the Webhook sends the POST request.



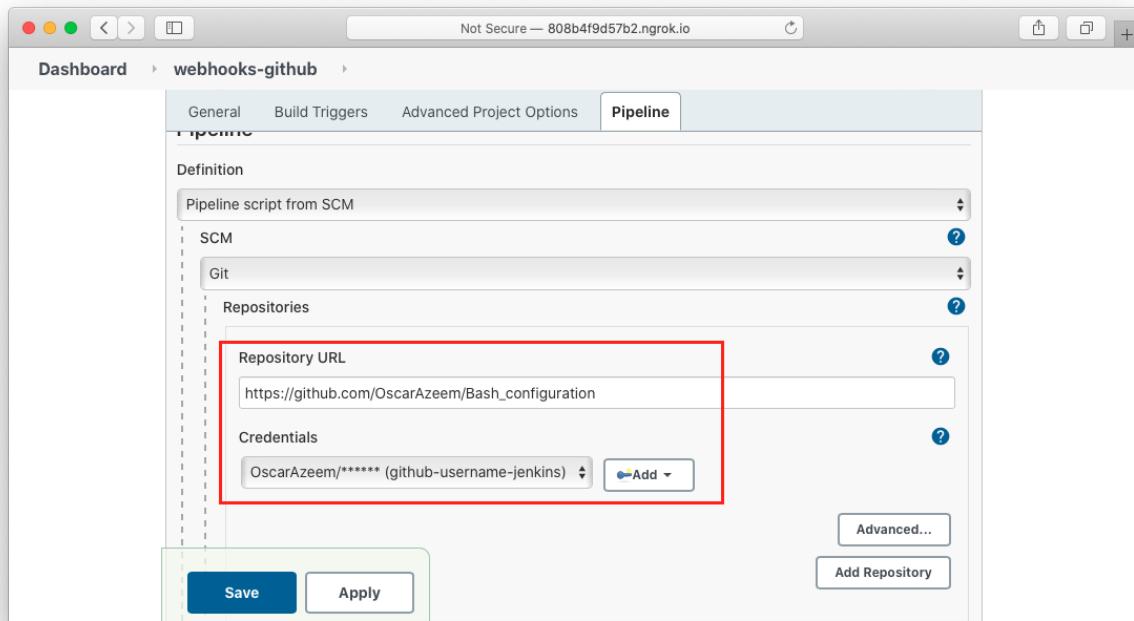
In the pipeline tab, choose: Pipeline script from SCM option to execute the steps declared in the Jenkinsfile for our project, select Git and paste your repository URL. Then we need to add our credentials. This step is not completely necessary but will enable GitHub to set a mark if our test passed or failed.



After clicking on add, it'll appear the credentials window. This time choose Username and Password. Write your GitHub username, but in the password text box, paste your GitHub API token, not your current password to access your GitHub account through the webpage.



Now your credential should be visible from the options.



Finally, we choose our branch to build the pipeline and the path for the Jenkinsfile. Click on save

Dashboard > webhooks-github >

General Build Triggers Advanced Project Options Pipeline

Branch Specifier (blank for 'any') Jenkinsfile_branch Add Branch

Repository browser (Auto)

Additional Behaviours Add

Script Path Jenkinsfile

Lightweight checkout

Save Apply

Our pipeline should look like the following image.

Jenkins

Dashboard > webhooks-github >

Back to Dashboard Status Changes Build Now Configure Delete Pipeline Full Stage View GitHub Rename Pipeline Syntax

Pipeline webhooks-github

Build triggered by GitHub events via Webhooks

Recent Changes

Stage View

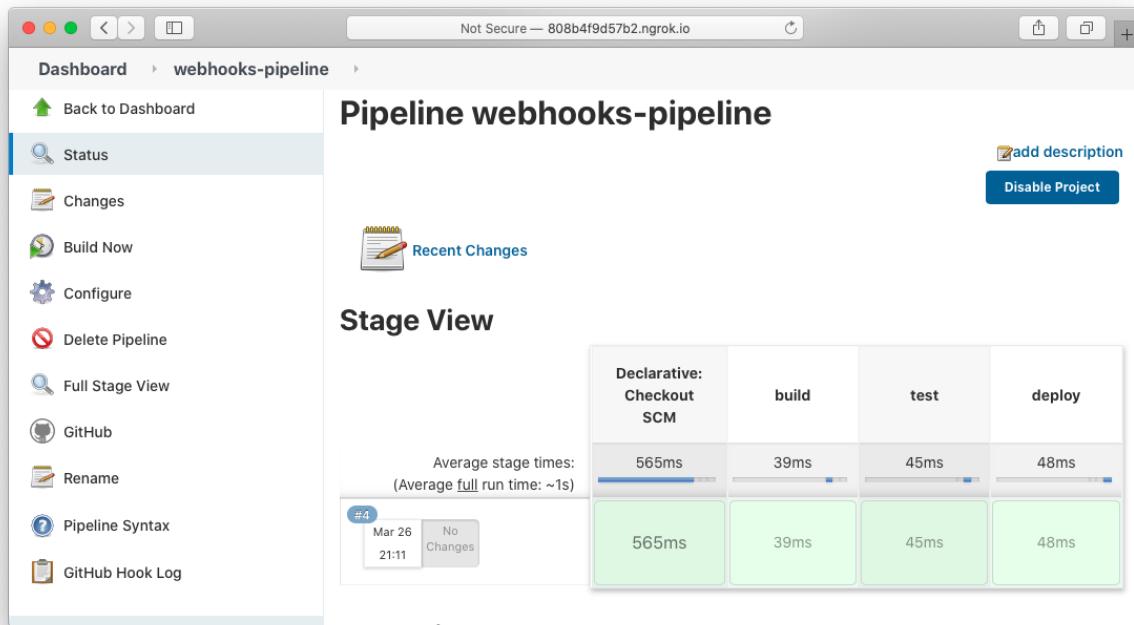
No data available. This Pipeline has not yet run.

Permalinks

edit description Disable Project

5. Pushing a commit to our GitHub repository and triggering the build

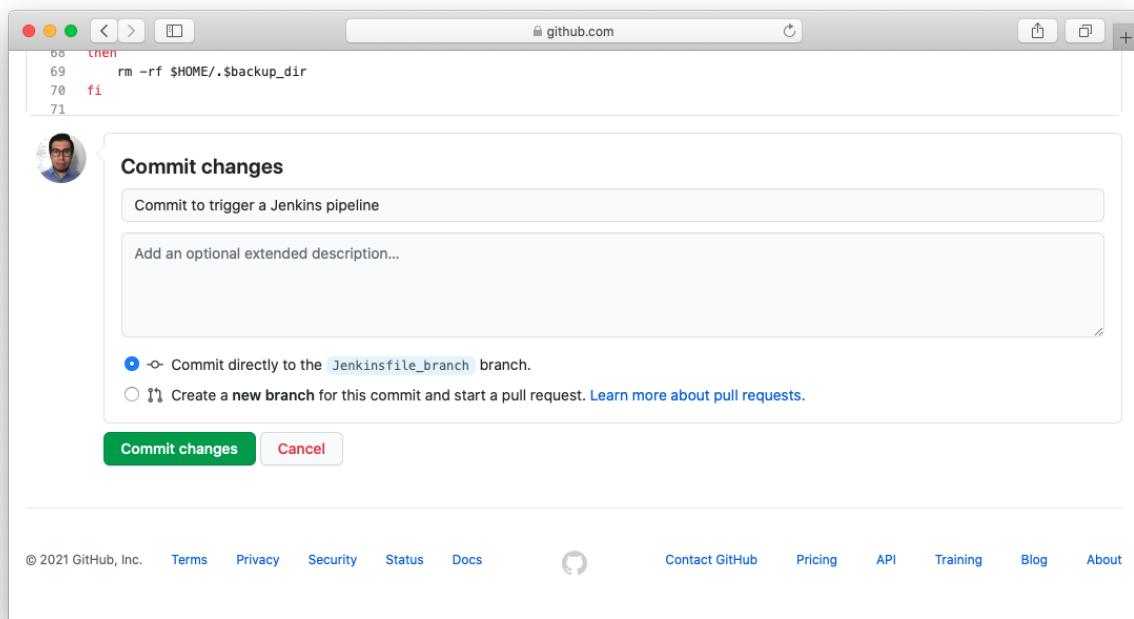
This process's main idea is to automate a Jenkins pipeline to run testing validations after some changes have been made in a remote repository. At this point, all the configurations have been realized successfully, so it's time to test it. Before committing a new change in our GitHub repository, we need to run manually at least one time our Jenkins job to be triggered by the webhook.



The screenshot shows the Jenkins interface for the 'webhooks-pipeline' project. On the left, there is a sidebar with various options: Back to Dashboard, Status (which is selected), Changes, Build Now, Configure, Delete Pipeline, Full Stage View, GitHub, Rename, Pipeline Syntax, and GitHub Hook Log. The main area is titled 'Pipeline webhooks-pipeline' and shows a 'Stage View' for the 'declarative: Checkout SCM' stage. The stage has four parallel steps: build, test, and deploy, each taking approximately 40ms. A summary table below shows the average stage times: Declarative: Checkout SCM (565ms), build (39ms), test (45ms), and deploy (48ms). A tooltip indicates that the last run was on Mar 26 at 21:11 with 'No Changes'.

Declarative: Checkout SCM	build	test	deploy
565ms	39ms	45ms	48ms
565ms	39ms	45ms	48ms

After our first run, let's go to our GitHub repository and create a new commit,



This commit will trigger the build automatically

A screenshot of a Jenkins dashboard for a project named "webhooks-pipeline". The left sidebar includes links for Changes, Build Now, Configure, Delete Pipeline, Full Stage View, GitHub, Rename, Pipeline Syntax, GitHub Hook Log, and Build History. The main area shows a "Recent Changes" section and a "Stage View" table. The table has four columns: Declarative: Checkout SCM, build, test, and deploy. The first row shows stage times: 668ms for checkout, 40ms for build, 42ms for test, and 37ms for deploy. The second row (build #5) shows 771ms for checkout, 41ms for build, 40ms for test, and 26ms for deploy. The third row (build #4) shows 565ms for checkout, 39ms for build, 45ms for test, and 48ms for deploy. A note indicates an average full run time of ~1s.

