**Continues Integration**

**Course Objectives:**

In this section you'll learn about the following:

* Continuous Integration
* Integration tools like GitHub Actions & Jenkins
* Setting up CI tools
* Run automated tests by pushing to central repository
* Run automated tests by sending pull-request to central repository
* Run automated tests inside Docker by a push
* Run automated tests inside Docker by a pull-request
* Quick overview in some Visual Studio Extension
* Where to go next

**Overview**

Throughout this course, you will know how important Continuous Integration to automate tests whenever code changes are introduced in team’s codebase. Also, you will be provided with project template (most of them has test files) so you don’t need to worry creating one. Always follow the lessons and instructions carefully.

Just to add, learning *Continuous Integration*in this course *will not require you so much coding*, but you may find *lots of configuration*(especially in Jenkins). If you find yourself struggling, don’t worry! It is due to learning something new and someday, you will find it easy to navigate.

To note also, the video above about *Git Graph (v1.28.0*) might not same with what current version you will use (as version updates and extension’s user interface can change time to time), but, a wise developer knows how to explore and compare. Feel free to check the documentation of [Git Graph](https://marketplace.visualstudio.com/items?itemName=mhutchie.git-graph) if you needed.

https://youtu.be/pV6\_2BEegjg

Good luck and happy learning!

#### Objectives:

* To know the concept of test automation
* To learn benefits of test automation
* To familiarize in different CI tools

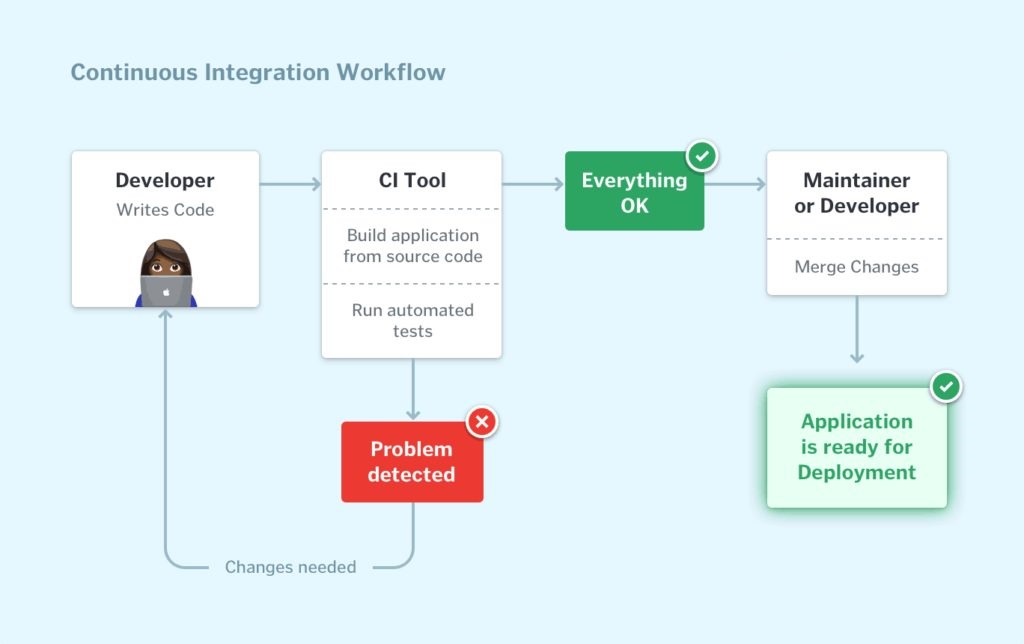
# Introduction to CI

**Continuous Integration (CI)** is the process of automating the build and test of code every time a team member commits changes to the shared repository.  When we say “integrate”, we’re pertaining to “combining” source codes, where developers can continuously introduce code changes in parallel.

CI promotes:

* Early detection of bugs
* Consistency for testing
* Time-efficiency (also for QA team as they’ll spend less time testing!)
* The goal of having fewer bugs in production
* Faster feedback for pull-requests if set

So imagine you’re pushing new code changes in GitHub repository, and for each ***push***, you get automatic feedback if the codes are safe or not by running unit tests in the background. Sounds cool, right? What if we say also that we can do it in **pull-request**– that for each creation, tests are run automatically and can visibly show the test result in pull-request details? Wow, amazing!

****

## Similarities in every CI Tools

After this lesson, you will learn first the simplicity of “**GitHub Actions**”, followed by the popular tool “**Jenkins**”. But whatever CI tool is introduced to you, remember what are the common structure of them all:

* ***All CI tools follows the concept of “workflow”.*** (workflow means description of the automation process that also has the step-by-step commands from start to finish)
* ***They all specify where to run the whole workflow*.**(ex. in your dedicated server, docker, or hosted environment provided by the CI tool)
* ***They will require you to mention the specific event that will trigger the automation.*** (ex. in every ‘push’ in central master branch)
* ***They will require you to give the step-by-step command flow.***(ex. you want the automation to compile your source codes, next is do some installation, then finally run unit testing).
* ***They allow you to import and use external, ready-made instructions.***

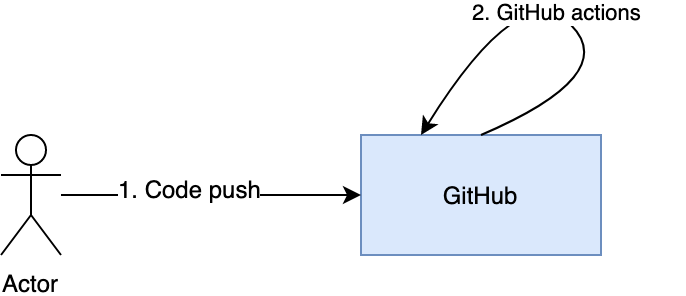
By the end of this course, you’ll be more flexible developer than you used to be. Have fun exploring!

#### Objectives:

* Learn what GitHub Actions is.
* Learn the anatomy of a workflow
* Plan the automation of your software development life cycle with GitHub Actions workflows.

# GitHub Actions

Now we know the concept of Continuous Integration. Let’s take a look first on how GitHub Actions is implemented.



**GitHub Actions** is an internal feature of GitHub. Since it resides inside GitHub, the setup made easy for us in continuous integration.

We can set it up that for every push in GitHub repository, GitHub will call "GitHub Actions" to perform automated testing. Test result will be return and show to GitHub repository.

Are you wondering where to see the result and how it is also done in pull-requests? Let's see by watching the below video:

**GitHub Actions** is a CI tool and has packaged scripts to automate tasks, and we instruct it how to do the testing using a workflow. Its workflow scripts adhere to .yml data format. (Note: You can locate the triggered workflows in your every GitHub repository via “Actions” tab.)

## Anatomy of GitHub Actions Workflow

Based on the diagram above, a workflow is a process that you set up in your repository. Through it, you can build, test, package, release, or deploy any project on GitHub. Workflow consists of jobs. Each job consists of steps and each step has an action.

Let's dig into the vocabulary of GitHub Actions.

* **Workflow**: a unit of automation from start to finish, including the definition of what triggers the automation, what environment or other aspects should be taken account during the automation, and what should happen as a result of the trigger.
* **Jobs**: a section of the workflow that groups the steps. Jobs are like “threads” in programming that runs concurrently. In the example above, we have only 1 job to be run – “build”. Every job requires at least one step.
* **Step**: represents one *effect* of the automation. We can add steps that can run commands, run setup tasks, or run specific action.
* **Action**: a piece of automation written in a way that is compatible with workflows. Actions can be written by GitHub, by the open-source community, or you can write them yourself!

In this point, you just read through a ton of information. Review all of it, and take some time to digest this stuff. It's not easy! This section is an advanced topic, so if you understood it, give yourself a big pat on the back. If you're struggling with the concepts, that's ok! Just take some time away from your computer and revisit this tab when you're feeling refreshed.

<https://youtu.be/vw21IbScTwU>

#### Objectives:

* Learn what GitHub Actions are, the types of actions, and where to find them.
* Plan the automation of your software development life cycle with GitHub Actions workflows.
* Create a container action and have it run in a workflow that is triggered by event to your GitHub repository.

# Scripting in YML

Now we will do how push and pull-request will work magically using YML file.

## Creating a Workflow

To create a workflow, you must create "***.github"*** folder first in your project. Inside it, you should create another folder named "***workflows***".  Add actions by having a new **.yml file** inside the ***.github/workflows*** directory. Note also that you can have multiple .yml files and these files will be read and run as independent tests. Moreover, the "**on**" parameter on each file will tell when it must be called, just like the below example:

name: Github Actions Example

on:

push:

branches:

- master

jobs:

build:

name: Build simple project

runs-on: ubuntu-latest

steps:

- name: Checkout project

uses: actions/checkout@v2

- name: Use Node.js

uses: actions/setup-node@v1

- name: Install dependencies

run: npm install

- name: Test project

run: npm test

REMEMBER when writing a ***.yml***file, as always, we uses tab for indentation?

Notice also the ***on:*** attribute. This is a trigger to specify when this whole workflow will run. In this example, it triggers when there is a push event to your repository.

A **workflow**must have at least one job. These are the pointers to remember:

* A **job**is a section of the workflow that will be associated with a **runner**
* A **runner**can be GitHub-hosted or self-hosted that can run the **job** on machine/container.

In above code, you are telling the **workflow**to run the job on ubuntu-latest.

Can you see that a **job**has **steps**to complete? Yes it has! In above example, there are **steps**that used both specific libraries and different terminal commands.

## What is "action" library?

 The power of **GitHub** **Actions**lies in access to actions written by the GitHub community. Here, we'll use two Actions officially written and supported by GitHub (you can browse more actions in GitHub Marketplace):

* ***actions/checkout@v2*** is used to ensure our virtual machine has a copy of our codebase. The checkout code will be used to run tests against. At this point of writing, the latest version is 2.
* ***actions/setup-node@v1*** is used to set up proper versions of Node.js since we can perform testing against multiple versions.

So, the next time you want to use action that is available from the GitHub Marketplace, you will type in .yml the keyword “**uses**” (in contrast to “**run**”) where you want to execute commands. You can check more action libraries in this [link](https://github.com/marketplace?type=actions).

## How to trigger Pull-request?

If we wanted to trigger the whole workflow in every pull request, we will use “**pull\_request**” event. (Note: Specifying branches are optional)

pull\_request:

branches:

- master

- develop

Yey! So simple, right? Once you apply it, you’re now a “Pull-stack developer”!

Just kidding!  ;)

To wrap-up our discussion, can you evaluate the GitHub actions by answering the ff. questions in your mind?

* ***Does it have the concept of “workflow”?***
* ***Did it specify which environment will use to run the whole workflow*? *Which syntax?***
* ***Did it require you to mention the specific event that will trigger the automation? How it is done in syntax?***
* ***How does it require the step-by-step command flow in syntax?***
* ***Does it allow to use external instructions?***

**Objectives:**

* To explore other git commands
* To apply continuous integration using GitHub Actions
* To do an auto-test when a commit is pushed to central repository

**Action thru Push**

So now, we already know that we can run unit tests automatically whenever codes are pushed to remote branch. We want you to explore other git commands. Go ahead choose one that you'll feel easy to code the logic. Let’s do it in an action!

Imagine your organization has inconsistent number of feature branches.

**TODO:**

**1.**Create new GitHub repository and copy your recent project “*git-commands*”. Whenever a commit is *pushed* from local to specific remote feature branch, a build should be triggered using GitHub Actions.

**2.**Once you finished the above, reconfigure your settings so that there will be automatic build whenever a commit is *pushed* from local to ANY remote feature branch. *(Hint: use wildcard)*

*Note:*

* *For this assignment, you do not need to have a pull-request.  Do the setup by having local and central repositories.*
* *For the #2, you can test the automation by pushing random codes to at least 2 feature branches (any name).*

Show your solutions by attaching screenshots. Good luck!

START WORKING ON THIS

**Objectives:**

* To apply continuous integration using GitHub Actions
* To do an auto-test when a commit is pushed to central repository
* To review in TDD

**Action thru PR**

For this challenge, we will continue the previous assignment by adding the following requirements:

* In your local, create 1 additional unit test and make it passed.
  + When done, push it in GitHub *feature*branch
* Configure settings to have pull-request restriction in develop and master branches.
  + When done, make a pull-request to *develop*branch
* Whenever a pull-request is opened, a triggered automated test's status should be seen in PR details.
* Make sure that a pull-request can’t ever be approved (even by admin) if the build is unsuccessful.

Show your solutions by attaching relevant screenshots. Good luck!

START WORKING ON THIS

**Objectives:**

* To apply continuous integration using GitHub Actions
* To debug action script in order to push successfully
* To run docker instead of default GitHub Actions' environment
* To do an auto-test inside docker when a commit is pushed to central repository

**Debug Docker in Action Push**

Remember *Docker*? Docker holds containers and container can be seen as a computer inside your computer. The cool thing about this virtual computer is that you can send it to your friends. And when they start this computer and run your code, they will get exactly the same results as you did without worrying of any software and hardware requirements!

This time, we will make the GitHub Actions use Docker environment (instead of the default environment that GitHub Actions uses) to build and run our test cases.

All you need to do is copy the repository "**survey-form-v2**" and try to figure out what’s wrong in***.github/workflows/action.yml****.*Focus on that file only when you're debugging.

GitHub Repository: <https://github.com/hh-kigcasan/survey-form-v2.git>

Push your changes in your personal repository and check if build will succeed. Once you’re done, attach your screenshot solution here and give yourself a tap!

START WORKING ON THIS

**Objectives:**

* To learn what Jenkins is
* To install tools in preparation for automation.
* To learn the purpose of each tool in automation.

**Jenkins**

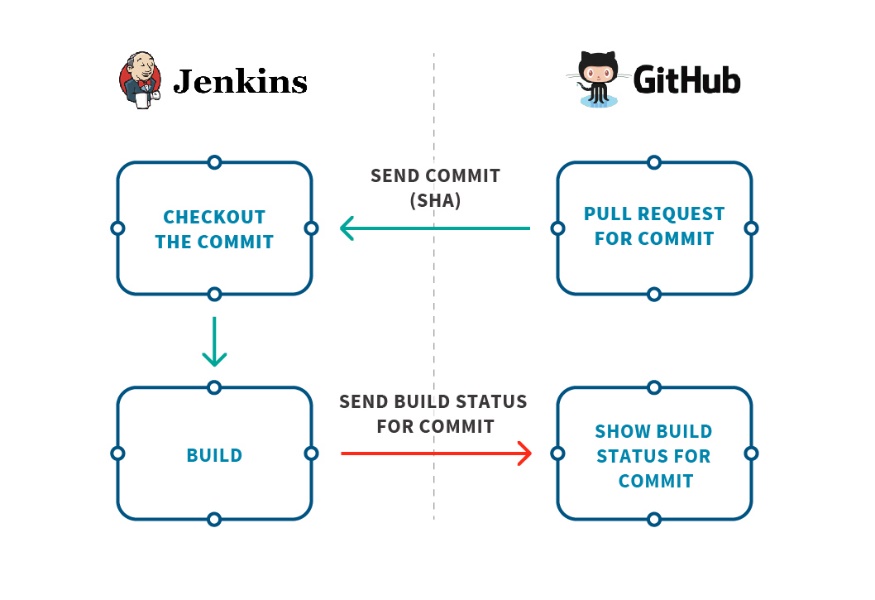
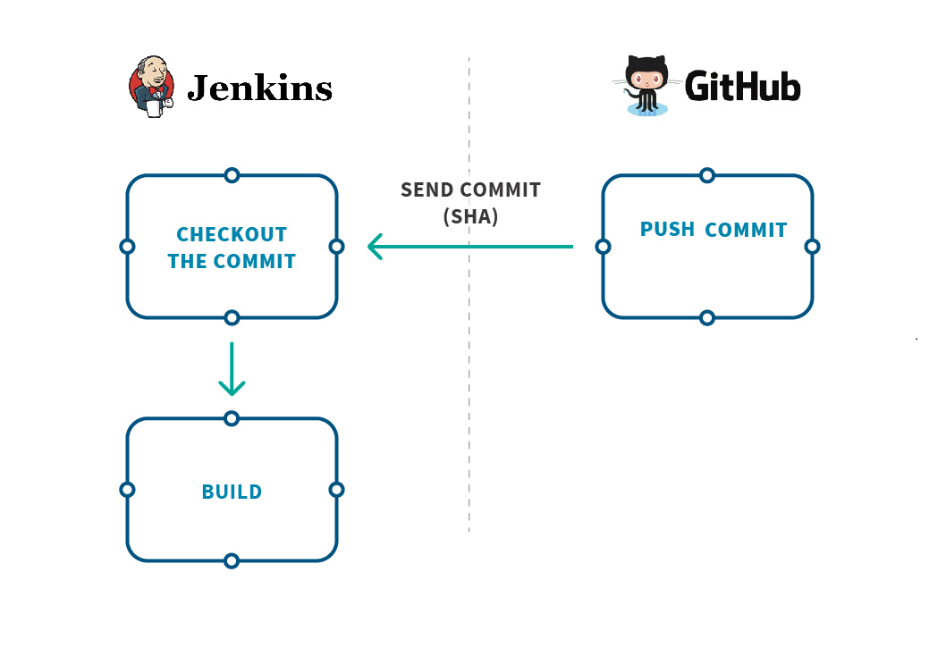
**Jenkins**is a free and open source automation server. It is widely used by tech industries since its first release in 2011. When you research, there are many CI tools out there, such as *Travis*, *CircleCI*, etc. but don't be overwhelmed! We will direct you to the correct learning path. **Jenkins**is very popular CI tool and surely your future employer will be amazed if you know it right!

Can you imagine that we can send pull-request or a push in GitHub, then **Jenkins**will do the test automation instead of *GitHub Actions*? In this case, GitHub is connecting to a CI tool externally.

Typically, when you're using **Jenkins**, you'll need it to host. This is unlike to GitHub Actions because it already resides in GitHub.

We will connect GitHub (as our version control system) and Jenkins (as continuous integration tool) to the rest of this chapter.

Analyze the below flow diagram and let's take a walkthrough in Jenkins portal!



<https://youtu.be/-julNRimApQ>

In the next lessons, we will teach you the basic tools before setting up how will be the continuous integration. You can compare also **Jenkins**and *GitHub Actions* in terms of configurations and portability.

Get ready to level up your skills, freshies!

Not all developers know how to set up continuous integration using Jenkins... and you're very lucky!

**Objectives:**

* To prepare primary tools to be used in next lessons
* To install and setup Jenkins in local computer
* To setup Ngrok in exposing local server to internet
* To generate GitHub personal access token to be used when Jenkins connects to repository

# Tool Preparation

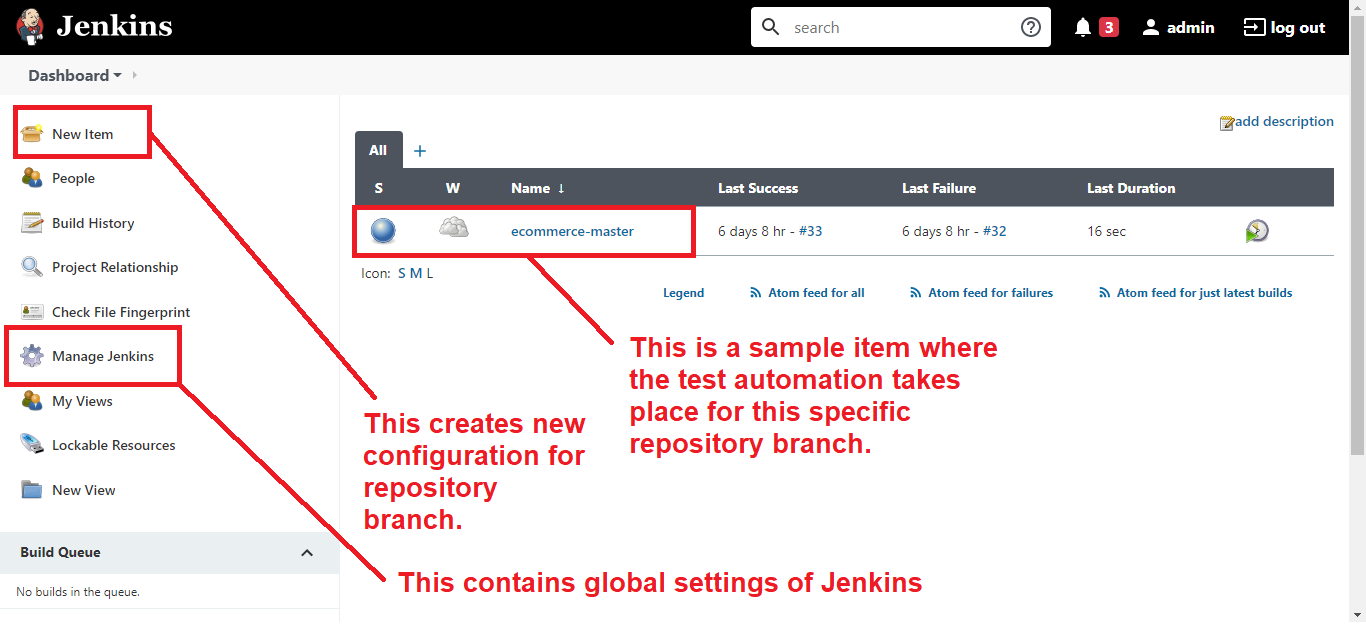
Before we teach how the integration will work, we want you to prepare the tools first.

## Setting up Jenkins

For this setup, we will host our Jenkins locally.

1. Download Jenkins using this link: <https://www.jenkins.io/>
2. As requirement, make sure you have Java version (ex. 8, 11) in your computer's environment variables (because Jenkins is built using Java language). After installation, restart your computer to take effect.
3. Go to your browser and navigate to your Jenkins server url. (default: [*http://localhost:8080*](http://localhost:8080/))
4. You will be prompted for login. For this case, you may look the suggested path OR check C:\Program Files\Jenkins\jenkins.err to copy the auto-generated password.
5. Install the recommended plugins. In case the plugins are still unsuccessful after many retries, don’t worry! You can skip for the meantime.
6. Once you logged in, go to Manage Jenkins from the left menu and click on Manage Users. Select the admin user and choose a new password. You will use the new password when you log in to Jenkins the next time.

Make sure you can login with your new password. Again, welcome to Jenkins portal! And for the meantime, we want you to focus more to the following:



## Setting up Tunnel using Ngrok

When we do continuous integration using Jenkins, GitHub will require us the URL of our Jenkins server. Unfortunately, [http://localhost:8080](http://localhost:8080/) is accessible only to our computer, and not to the internet. To solve this, we will use a free software named “Ngrok” to expose our local server in super minimal effort! Cool, huh!? These are what we need to do:

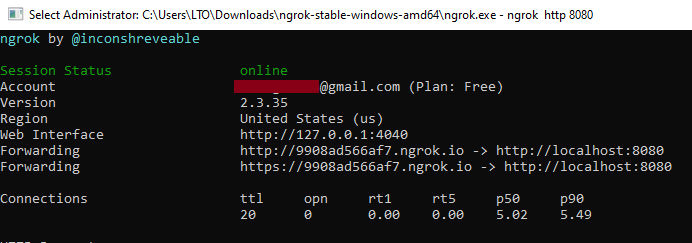
1. Create an account in [https://ngrok.com](https://ngrok.com/)
2. Go to your dashboard and follow instructions: <https://dashboard.ngrok.com/get-started/setup>
3. Make sure you downloaded the file.
4. Run the ngrok.exe file as administrator and manually put the following:

ngrok authtoken <token>

ngrok http 8080

Since we wanted to expose our Jenkins server with port 8080, using the generated link (http://<hash>.ngrok.io), everyone can visit the portal in their own browser as long as the ngrok terminal is up. Wow!

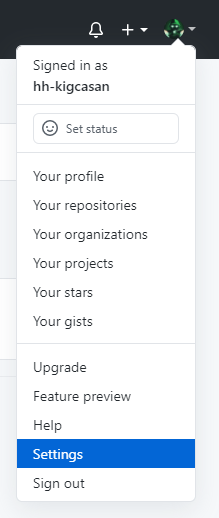
Note: To copy link, highlight the given HTTP url and Ctrl+Shift+C.



## Generating GitHub Access Token

Later when we will connect our Jenkins server to our GitHub account, we have to provide GitHub access token to authenticate and authorize, in order to permit Jenkins to access some repository information.

Let’s prepare it by navigating first to Settings:



From the left menu, click Developer Settings > Personal access tokens, and from there, generate new token with checked scope “repo”.

Congrats! You now reached the end of tool preparations! Looks easy, right? This is just only the start of setting up. You'll know more the difference of Jenkins to GitHub when you dive to its practical usage.

Let’s now move on how to make test automation using Jenkins!

#### Objectives:

* To explore Jenkins environment and how CI works through it
* To learn how auto-build triggers for every push event
* To utilize Jenkins plugins in achieving the automation

# Setup Jenkins Push

In this section, you'll find yourself configuring a lot than what we did in previous CI tool... embrace challenge!

When using **Jenkins**, we’re not looking anyway in the Actions tab in our GitHub repository. Instead, we’re looking externally that is located in [http://localhost:8080](http://localhost:8080/) (as we setup). Same as GitHub Actions, we can also setup our continuous integration and expect that there will be a trigger whenever a commit is push to remote repository.

Watch how is the configuration for push event:

Go ahead and configure the following:

## GitHub Webhook

When you visit your repository and navigated to Settings, can see you the Webhook menu in the left side? Webhook is in-charge in informing external servers whenever an event is triggered (for example, developer pushes new code, or sent a pull-request, etc). Since we are not using GitHub Actions anyway to execute triggers, we are required now to use Webhook.

Any information that Webhook wants to send will be routed to the given Payload URL, where in our case, our tunneled Jenkins portal, appended with “/github-webhook/”. We can see also that it has many options in which events we wanted it to trigger. But for now, let’s choose “Only the push event”.

NOTE: We can also monitor here if GitHub webhook delivers our Payload URL successfully or not.

## Jenkins Global Settings

We will save now our GitHub credential in global settings. Just go to Jenkins Dashboard > Manage Jenkins > Configure Systems > Scroll down until you see “GitHub” section. Remember the GitHub Access Token we created earlier? Put it as your credential and test if it can connect successfully. Hit ‘Apply’ to save the settings.

## Creating Jenkins Item

Now, we will create an item that will be responsible in receiving the details from GitHub webhook. Once webhook routed successfully to specific item, item will execute a build. Remember how we created a workflow in the GitHub Actions? This concept is not different to Jenkins in terms of executing tests.

Steps:

1. Let’s click “New item” from Jenkins Dashboard to create. Populate the item name field (You may use format: <repository name>\_<branch>) and choose “Freestyle project”. Hit “Ok” button.
2. Configure our freestyle project -- same setup in the video.

Here are the important settings:

* + **General**> GitHub project – make sure to populate
  + **Source code Management**> Git – will be used to checkout the repository during the Jenkins build later. Make sure to populate the required fields. Specify also which branch will be accepted for Jenkins build only whenever the GitHub Webhook sends data.
  + **Build Triggers**> GitHub hook trigger for GitSCM Pooling – make sure to check
  + **Build >**Execute shell – this will contain the steps in our workflow that we wanted the Jenkins build do.

Notice that we can click the question mark button to know the field's purpose.

Once you configure it all, you’re on the right track! Go ahead and try some push in your remote codebase and check your Jenkins portal if it builds automatically!

To wrap-up our discussion, can you evaluate Jenkins by answering the ff. questions in your mind?

* Does it have the concept of “workflow”?
* Can we specify which environment will use to run the whole workflow?
* Did it require you to mention the specific event that will trigger the automation? How it is done?
* How does it require the step-by-step command flow in syntax? Which part of settings?
* Do you know that it allows to use external instructions called plugin?

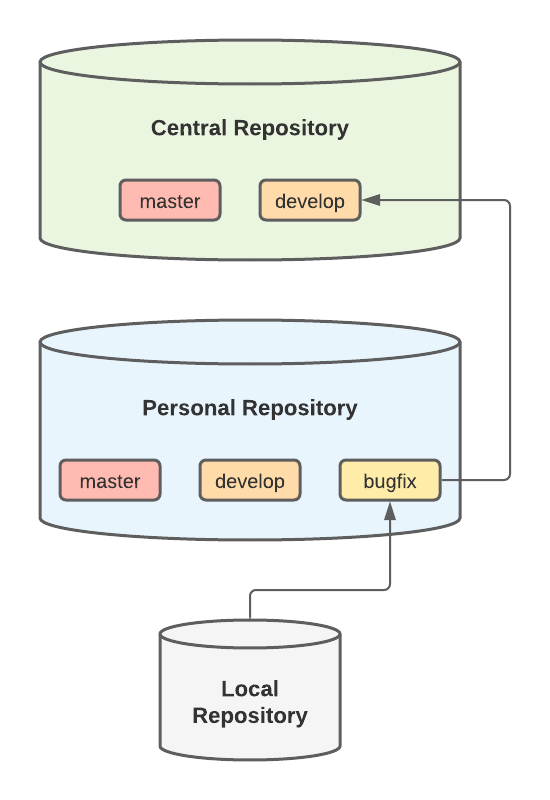
<https://youtu.be/IW43VaFoBEg>

**Objectives:**

* To setup Jenkins locally
* To configure GitHub settings in triggering Jenkins
* To appear the build status in portal if triggered a push event

**Trigger thru Push 1**

Setup Jenkins and create a repository that is configured so that whenever a commit is pushed from local to remote repository, an automatic build will be triggered in Jenkins. To save time, we will reuse your *survey-form-v1* repository (from "Pull Request 3" assignment) in order to do this. Remember how is the repository design?



For this task, we will have simple practice how push event and automated testing will work. We'll not be concern in Central Repository and specific codes that we will push for the meantime. Hence, your task is to setup Jenkins so that you'll have automation in personal/*bugfix*branch*.*

Requirements:

* Should have Jenkins item named “*<repository name>*\_<*branch>*”.
* Make sure that build will start automatically in your Jenkins server [http://localhost:8080](http://localhost:8080/) by having any code changes pushed from local to personal branch.

Once you’re done, attach your screenshot solutions here. Good luck!

*Tip: If you encounter "Cannot run program in sh" error in Jenkins build logs, it means you're using shell commands but no shell path specified that will be used. To solve, set the Manage Jenkins > Configure System > Shell > Shell executable input field with your 'sh' path (ex. C:\Program Files\Git\bin\sh.exe).*

START WORKING ON THIS

**Objectives:**

* To run test cases in docker inside Jenkins server
* To configure GitHub settings in triggering Jenkins
* To appear the build status in portal if triggered a push event

**Trigger thru Push 2**

In this assignment, we will build and run test cases inside **Docker**. We will utilize our Jenkins server to perform the automation testing. For the meantime, we will not be concern to specific codes and branch name.

Do the following steps:

1. Create a new remote **central**repository in GitHub from <https://github.com/hh-kigcasan/survey-form-v2.git>
2. Create and configure **personal**repository’s settings specific to branch *feature/abc*.
3. Setup you Jenkins by creating new item named “survey-form-v2\_feature-abc”.
4. Test the automation by pushing codes from ***local feature/abc*to *personal feature/abc*** branch. Make sure that Jenkins is triggered only (not including GitHub Actions).

Once you’re done, attach your screenshot solutions here. Good luck and you can do it!

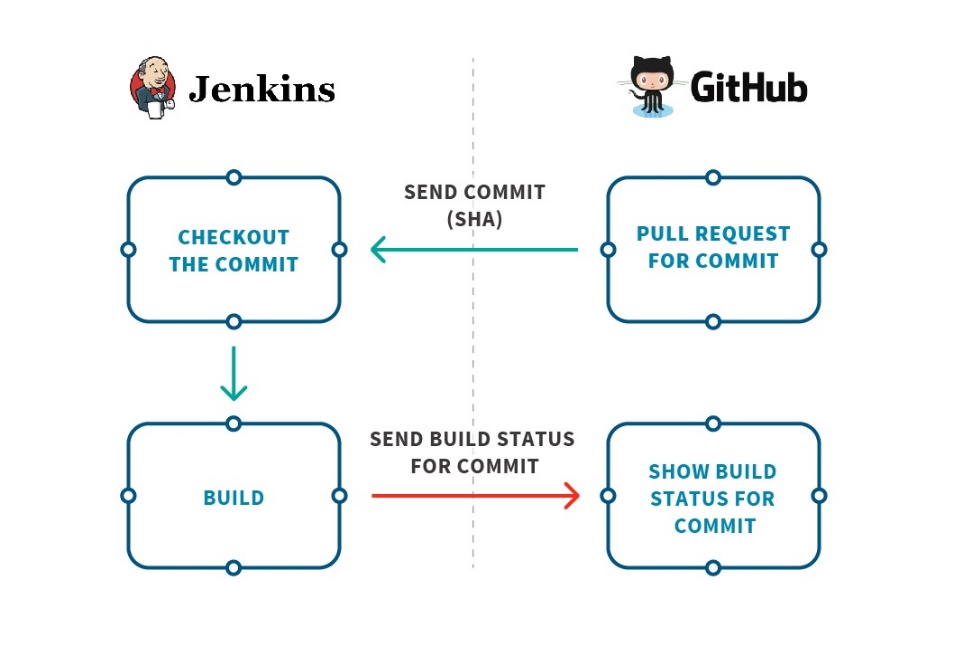
START WORKING ON THIS

**Overview:**

* To explore Jenkins environment and how CI works through it
* To learn how auto-build triggers for every pull-request event
* To utilize Jenkins plugin in achieving the automation

**Setup Jenkins PR**

Jenkins is so powerful CI tool and its plugins extend the overall functionality. Before we discuss the steps on how to configure the automation, let’s analyze first the big picture:

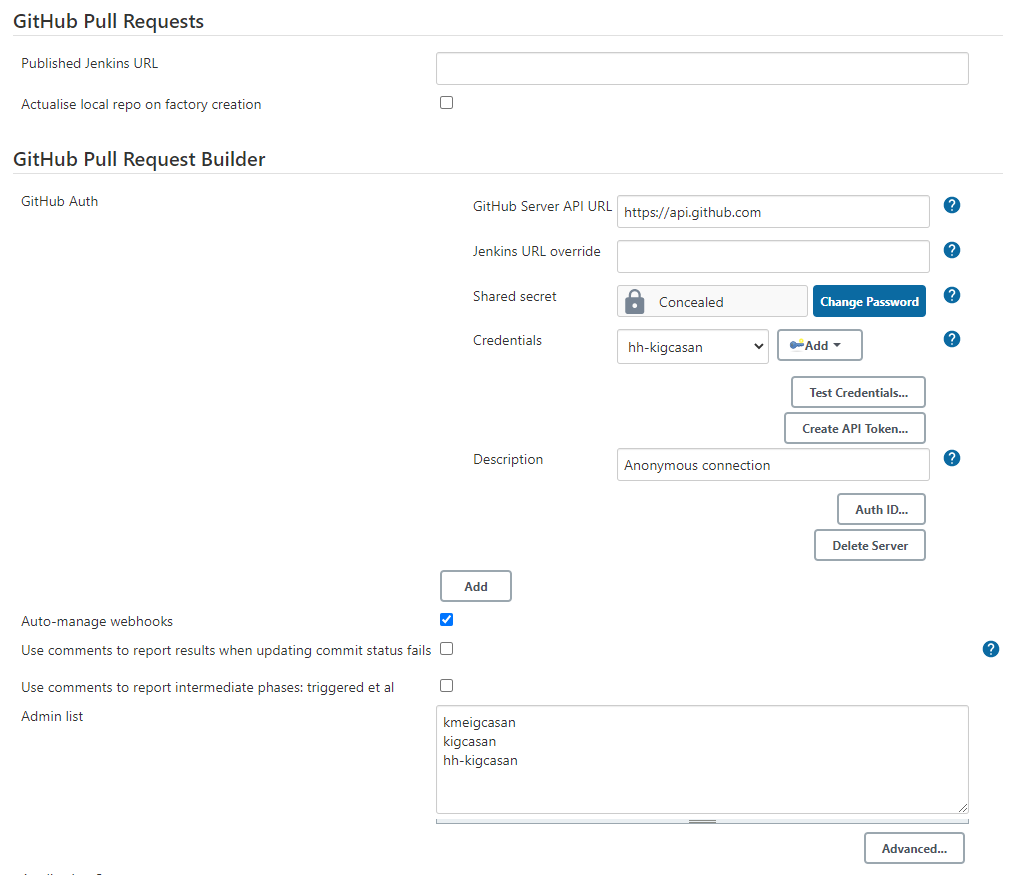


Based on the flow, whenever a **pull-request** is opened, GitHub will send a request to Jenkins to test the proposed code changes to destination branch. What Jenkins will do is it will get the whole copy of destination branch’s codes, add the new codes (from pull-request), then tests if it will fail or not. After this process, Jenkins will return a build result to GitHub which will reflect on the pull-request. Hooray! Team reviewers can see now if the pull-request is acceptable to merge or not.

Let’s watch the video below to see it in an action:

Jenkins alone can’t achieve delivering of build report. Therefore, we will use a plugin to manage the flow. Go ahead in Manage Jenkins > Manage Plugins > Install “**GitHub Pull Request Builder**”.

Once you’re done installing, go to Manage Jenkins > Configure System, and look for GitHub Pull Request Builder section:

****

Notice that we have default GitHub Server API URL, and to connect with it, use your GitHub Access Token as credential. Make sure that connection is stable, and auto-manage webhooks is checked. For the Admin list, type the GitHub usernames which will be recognized by the plugin when executing the workflow.

When you create a new **Jenkins item**, take note that you need to check the following:

✓**Build Triggers** -> GitHub Pull Request Builder

* GitHub API credentials: [https://api.github.com](https://api.github.com/)
* Admin list: <list of GitHub usernames>
* Use GitHub hooks for build triggering (checked)
* Build every pull request automatically without asking (checked)

✓**Build Environment** -> Set GitHub commit status with custom context and message (Must configure upstream job using GHPRB trigger)

Now as you learned how Jenkins accept testing from GitHub pull-request, do not forget to ***add new webhook*** and ***branch restriction in GitHub***, as well as setting the ***Branch Specifier*** in Jenkins item, etc.

Have fun experimenting!

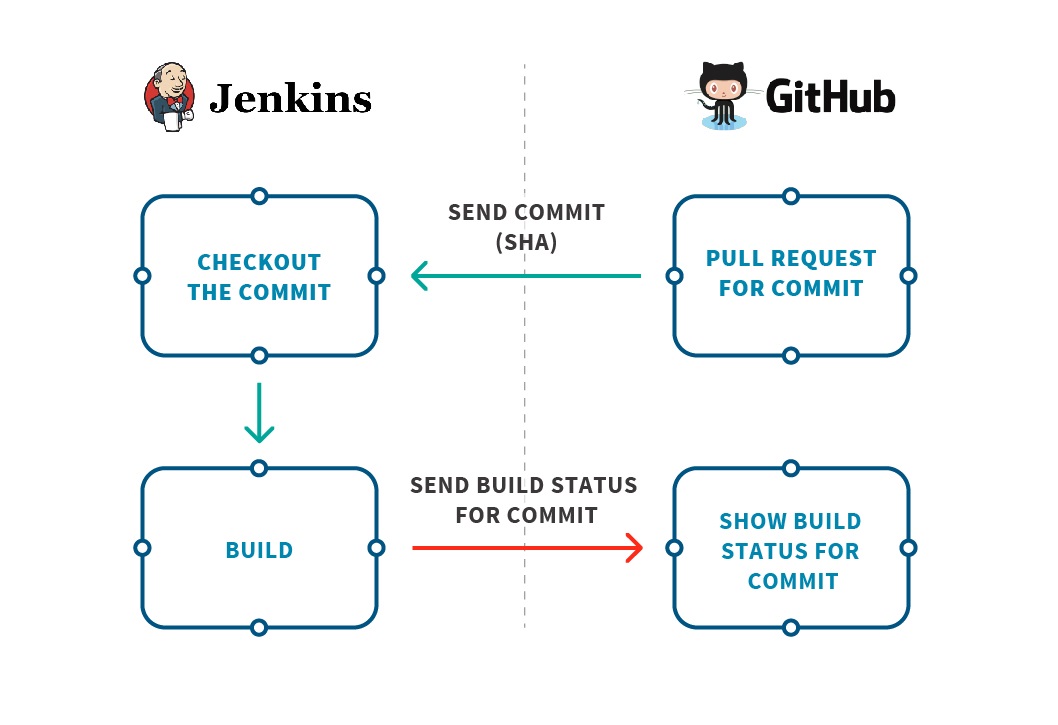
<https://youtu.be/mir6yCQ8fP0>

**Objectives:**

* To do an auto-test when a pull-request is opened from personal to central repository
* To run testing using docker inside Jenkins server

**Trigger thru PR**

For this assignment, we will reuse our repository from previous assignment, but now with *develop* branch. Configure its settings so that whenever a pull-request is opened (from ***personal/feature* to *central/develop*** branch), only Jenkins build will be triggered.



**To-do’s:**

* Create Jenkins item named “survey-form-v2\_develop-pr”
* Make code changes from personal/feature and have PR to central/develop. See if the route is correct by having automatic build to Jenkins item “survey-form-v2\_develop-pr”
* Make sure that tests are run inside Docker.
* Each build’s description in Jenkins portal should be in generated format: “Pull-request #<id>: <title>“
* Status returned by Jenkins should have visible caption “Build success”, “Build failed”, “Build error” (depends on the status) in GitHub pull-request details.

Note: In order to do this, you may try to work in first 3 to-do items before going to last 2 items. Good luck and kindly attach your solutions in image format.

# Wrap-up

Congrats! You made it at the end! What you learned now is stronger than before!

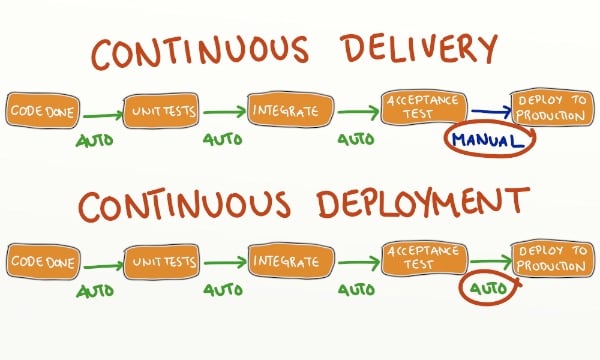
Remember that we introduced to you the advance Git concept -- the branching, pull request, resolving merge conflicts? Don't forget these because a lot of times, you'll deal with it in daily routine in your work team.

And to summarize also, we tackled the Continuous Integration and its tools. You can now differentiate the two tools in terms of setting it up and the coding part. These are only two examples of CI tools that you can find when you research, but whatever may your future employer uses (even none of the two), you now have the basic concepts and the grasp to configure it.

## Some tips

**CI/CD**

When you do some research about continuous integration, you may see often times the terms "Continuous Deployment", "Continuous Delivery". Deployment means the act of updating environment's codes (ex. production) based from the latest developed codes. To give you quick overview, here's the big picture.



The image above just says the manual or automatic approval of deployment. And that's what it is all about!

**Git Branching Model (Git Flow)**  
From this course, we talked about master, develop, feature branches. When you're in a working environment soon, it's important to observe how they use branching. Some companies might have their own style (for example, if bugfix, they'll direct push to develop branch). Just keep the learnings you got from this course and be a flexible developer.

In the past modules, only master, develop, feature, bugfix branches were discussed. You will find this as good resource to leverage your learning about branching:  
<https://nvie.com/posts/a-successful-git-branching-model/>

**Jenkins (Advance)**

If you dig through Jenkins, a lot of times you'll find that it uses scripting too. If GitHub Actions has .yml workflow, Jenkins has Jenkinsfile written in Groovy language. When you're using Jenkinsfile also, you'll be required to specify stages (or steps) in running the whole workflow. If you made it, you will see pipelines (or swimlanes that let you visualize at which stage your automated build is at current time).

So the next time you encounter again Jenkins as CI tool used by your employer, be prepared for this advance concepts!

Good luck and keep up the good work!

# Git & CI Course Evaluation

Your feedback helps us to know how we can improve our training courses. Please give your honest feedback for the questions in this Evaluation form:

<https://docs.google.com/forms/d/e/1FAIpQLSdUb-Tix4GK62lHofOnwcn9dZ1DtAc2CasM6sdmY4b7_QgOfQ/viewform>