



KodeKloud

© Copyright KodeKloud

Follow us on <https://kodekloud.com/> to learn more about us.



Project Status Meeting - 1

Project Status Meeting - 1

Priority	Task	Assigned	Status	Comments/Issue
0	Understand Requirement	Alice	Not started	
1	Unit Testing	Alice	Not started	
2	Code Coverage	Alice	Not started	
3	Containerization	Alice	Not started	
4	Kubernetes Dev Deployment	Alice	Not started	
5	Dev Integration Testing	Alice	Not started	
6	Manual Approval	Alice	Not started	
7	Kubernetes Prod Deployment	Alice	Not started	
8	Prod Integration Testing	Alice	Not started	

Project Status Meeting - 1

Priority	Task	Assigned	Status	Comments/Issue
0	Understand Requirement	Alice	In Progress	
1	Unit Testing	Alice	In Progress	
2	Code Coverage	Alice	In Progress	
3	Containerization	Alice	In Progress	
4	Kubernetes Dev Deployment	Alice	Not started	
5	Dev Integration Testing	Alice	Not started	
6	Manual Approval	Alice	Not started	
7	Kubernetes Prod Deployment	Alice	Not started	
8	Prod Integration Testing	Alice	Not started	



Understanding NodeJS Application

NodeJS

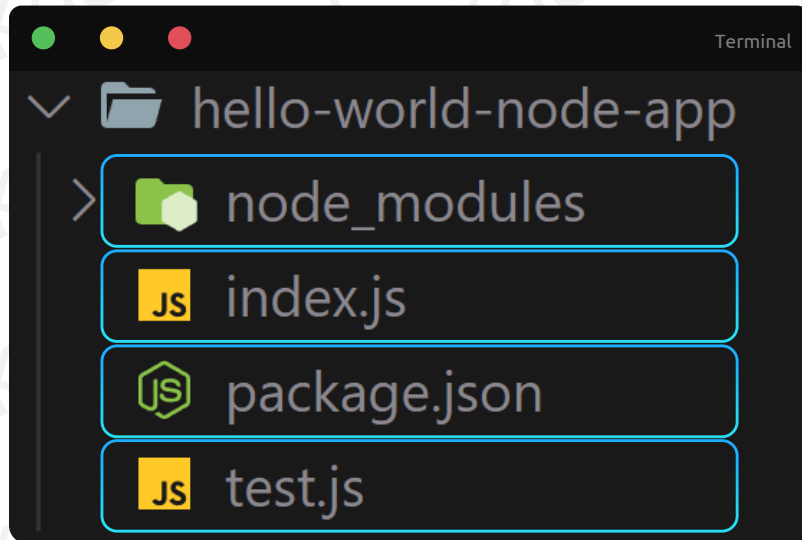


© Copyright KodeKloud


Let's take a brief look into Node.js. The demo application to understand Github actions is developed using Nodejs . In a later module, we'll utilize the Node.js application to develop a customized GitHub action pipeline.

1. Node.js is an open-source runtime environment that enables developers to execute JavaScript code outside of web browsers.
2. With Node.js allows developers to create both front-end and back-end applications using JavaScript.
3. Nodejs is built on the Chrome V8 JavaScript engine. This makes it possible to build back-end applications using the same


NodeJS



```
Terminal
hello-world-node-app
> node_modules
  index.js
  package.json
  test.js
```

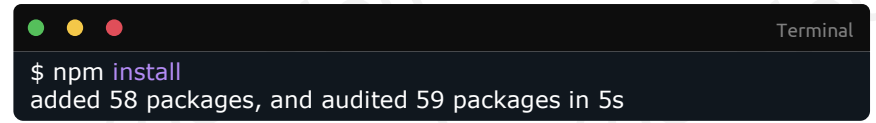


```
Terminal
$ curl localhost:3000/hello
Hello World!
```




```
Terminal
$ node -v
v18.16.0


$ npm -v
9.8.1
```



```
Terminal
$ npm install
added 58 packages, and audited 59 packages in 5s
```



```
Terminal
$ npm test
Testing is successful
```



```
Terminal
$ npm start
App listening on port 3000
```

© Copyright KodeKloud

Let's take a brief look into Node.js. The demo application to understand Github actions is developed using Nodejs . In a later module, we'll utilize Node.js to develop a customized GitHub action.

Node.js is an open-source runtime environment that enables developers to execute JavaScript code outside of web browsers. Node.js allows developers to create both front-end and back-end applications using JavaScript.

Nodejs is built on the Chrome V8 JavaScript engine. This makes it possible to build back-end applications using the same JavaScript programming language you may be familiar with.

Node.js can be installed on all major platforms, including Windows, macOS, and various Linux distributions. After installation, you can check the installed versions by typing `node -v` and `npm -v`.

What is npm?

npm stands for "Node Package Manager." It is a package manager for JavaScript and Node.js applications. It is used to discover, share, distribute, and manage the various packages, libraries, and dependencies that JavaScript developers use when building web and server-side applications.

Npm gets installed as part of the Nodejs installation.

Let's explore a sample Node.js project to learn how to run tests and execute nodejs apps. This is a basic, minimal Node.js application that displays 'Hello World.'

A `package.json` file in a Node.js project is a metadata file that includes essential information about the project, like its name, version, dependencies, and scripts. It is used for dependency management, enabling you to specify which external packages your project relies on and their versions.

To install the dependencies defined within `package.json`, we run `npm install`.

Once install is successful a `node_modules` directory gets created which contains all the external JavaScript modules and packages that your project depends on.

Finally we have the `index.js` file where the actual business logic exists and a `test.js` file is used to define the test cases.

npm test command is commonly used to run tests for a JavaScript project. This command runs all the tests defined in test.js file

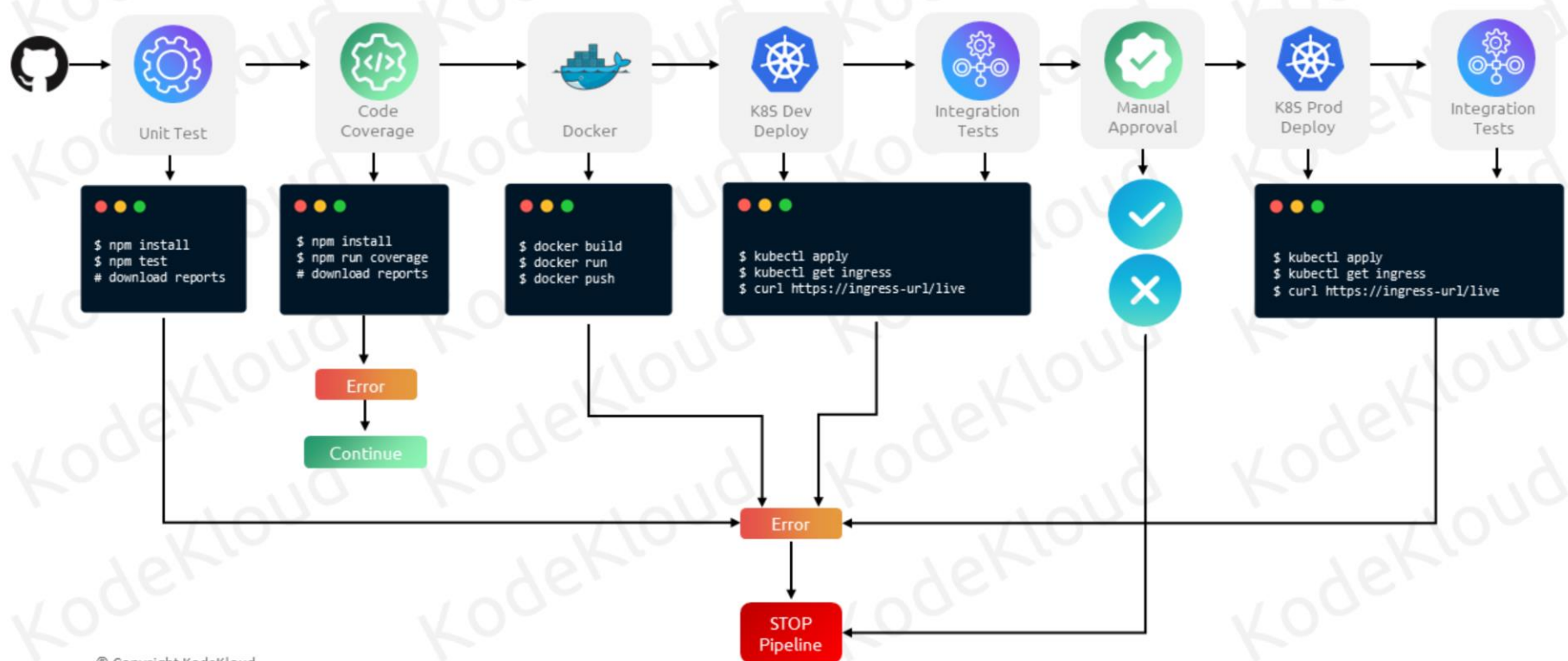
In a Node.js project, npm start is a common command used to start your application.

Once the application is started we can access it on the browser.



Understanding DevOps Pipeline

Understanding XYZ Team DevOps Pipeline





GitHub Action Expressions

Expressions

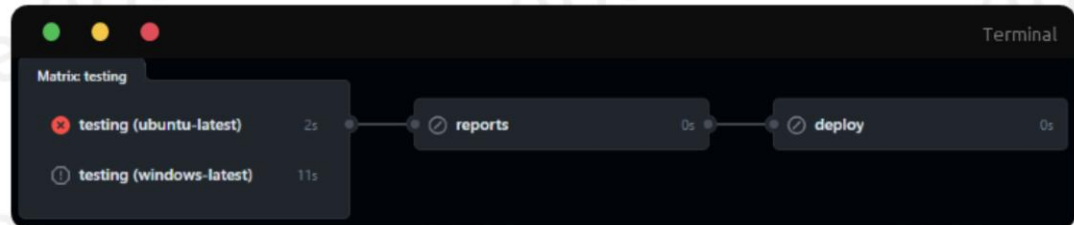
```
on: push

jobs:
  testing:
    strategy:
      matrix:
        os: ['windows-latest', 'ubuntu-latest']
    runs-on: ${ matrix.os }
    steps:
      - name: Testing on Ubuntu
        run: |
          export apikey=$3cuR3-t0k3N
          echo "Running Tests .. . . . ."

      - name: Testing on Windows
        run: |
          Set-Variable -Name "apikey" -Value "$3cuR3-t0k3N"
          echo "Running Tests .. . . . ."

  reports:
    needs: testing
    runs-on: ubuntu-latest
    steps:
      - name: Upload Report to AWS S3
        run: echo "Uploading reports .. . . . ." && exit 1

  deploy:
    runs-on: ubuntu-latest
    needs: reports
```



```
Terminal

✓ Testing on Ubuntu
1 ▶ Run export apikey=$3cuR3-t0k3N
5 Running Tests .. . . . .

✓ ✗ Testing on Windows
1 ▶ Run Set-Variable -Name "apikey" -Value "$3cuR3-t0k3N"
5 /home/runner/work/_temp/8153837b-b372-4cc5-ab68-c3f1fe8de8ca.sh: line 1: Set-Variable: command not found
6 Error: Process completed with exit code 127.
```

© Copyright KodeKloud

Before diving into expressions, let's examine this sample workflow.

This workflow comprises three jobs: testing, reports, and deploy.

The testing job contains two steps, which execute tests on both Windows and Ubuntu machines while configuring an environment variable.

The reports job contains a single step that uploads reports to an S3 bucket and then terminates.

For now, let's ignore the deploy job.

When the workflow is executed, it is bound to encounter a failure in the testing job. This is because the "Set-variable" command cannot be executed on an Ubuntu machine, leading to the automatic skipping of subsequent steps and jobs.

This undesirable error calls for a solution, particularly since testing on multiple operating systems is a common necessity. We will address this challenge using GitHub Actions expressions.

Expressions

Use expressions to programmatically execute **JOBS** and **STEPS** based on conditions

if

steps:

```
- name: Testing  
  if: <expression> || <expression>  
  run: ./script.sh
```

jobs:

```
Deploy:  
  if: github.ref == 'refs/heads/main'  
  steps:
```

continue-on-error

steps:

```
- name: Testing  
  continue-on-error: true  
  run: ./script.sh
```

jobs:

```
Deploy:  
  continue-on-error: false  
  steps:
```

contexts

github

env

vars

job

jobs

steps

runner

secrets

matrix

needs

inputs

strategy

GitHub Actions expressions are used to dynamically configure workflows to execute **JOBS** and **STEPS** based on conditional logic.

Two primary conditions come into play: "if" and "continue-on-error."

The "if" condition empowers you to control whether a particular step or job should proceed based on a defined condition. The if condition is typically used within a job's or step's configuration to determine its execution.

An expression can be any combination of literal values, references to a context, or functions.

As seen earlier, contexts are a set of pre-defined objects or variables containing relevant information about the environment, events, or other data associated with a workflow run. You can use contexts to access information about the workflow, such as the branch that was triggered the workflow, the commit that was pushed, or the runner details where the workflow is getting executed.

These are some of the most commonly used GitHub Actions contexts

The continue-on-error option in GitHub Actions allows you to specify whether or not a job should continue running even if a step fails. By default, if a step fails, the job will stop running. However, if you set continue-on-error to true, the job will continue running even if the step fails.

The continue-on-error option can be set at the job level or the step level. If you set it at the job level, it will apply to all steps in the job. If you set it at the step level, it will only apply to that specific step.

Expressions

```
on: push

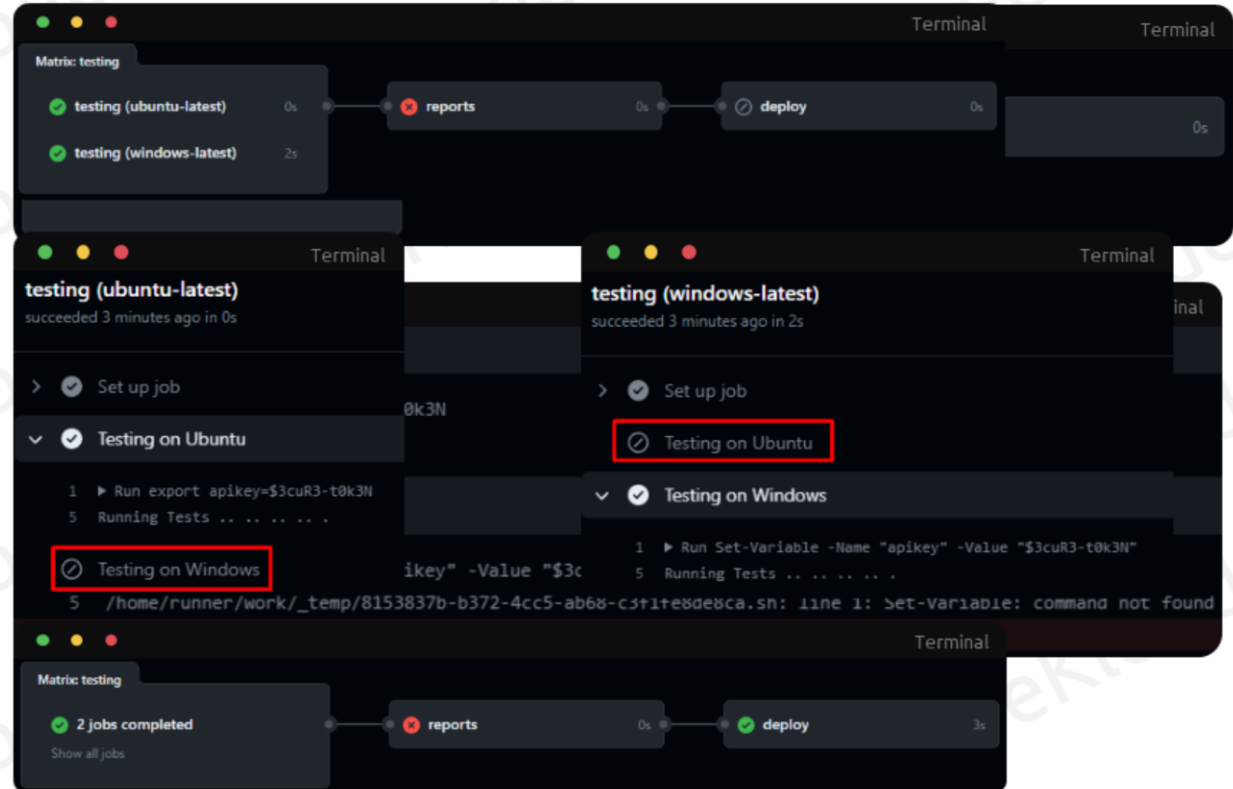
jobs:
  testing:
    strategy:
      matrix:
        os: ['windows-latest', 'ubuntu-latest']
    runs-on: ${{ matrix.os }}
    steps:
      - name: Testing on Ubuntu
        run: |
          export apikey=$3cuR3-t0k3N
          echo "Running Tests .. . . . ."

      - name: Testing on Windows
        run: |
          Set-Variable -Name "apikey" -Value "$3cuR3-t0k3N"
          echo "Running Tests .. . . . ."

    reports:
      needs: testing
      runs-on: ubuntu-latest

    steps:
      - name: Upload Report to AWS S3
        run: echo "Uploading reports .. . . . ." && exit 1

    deploy:
      runs-on: ubuntu-latest
      needs: reports
```



© Copyright KodeKloud

Lets revisit this example and make changes to run the testing job successfully.

We will introduce "if" conditions to both steps within the testing job. These conditions will verify if the runner's operating system matches the specified literals, either Linux or Windows. If the condition evaluates to true, the step will be executed; otherwise, it will be skipped.

With these "if" conditions in place, the testing job will run successfully. However, the deploy job remains skipped due to

errors encountered in the upload reports job.

Now, let's consider the reports job as optional, and its outcome should have no impact on other steps or jobs within the workflow. To achieve this, we can disregard its result by adding a "continue-on-error: true" condition at the job level. Consequently, when we execute the workflow, the deploy job will proceed even if the report job fails. This behavior is attributed to the "continue-on-error" condition.

The continue-on-error option can be a useful way to allow jobs to continue running even if some steps/other job fail. However, it is important to use it carefully, as it can lead to missed errors.

Expressions (Status Check Functions)

Use expressions to programmatically execute **JOBS** and **STEPS** based on conditions

if

steps:

- name: Build
run: npm build
- name: Test
run: npm test
if: success()

steps:

- name: Always Run
run: echo "step always runs"
if: always()

success()

failure()

cancelled()

always()

© Copyright KodeKloud

GitHub Actions provides various status check functions and expressions that you can use in your workflow to check the status of different aspects of your workflow, such as jobs, steps, and contexts. These functions help you make decisions and control the flow of your workflow based on certain conditions. Here are some commonly used status check functions:

success() and failure():

These functions are used to check the success or failure status of a previous job or step. You can use them in conditional statements (e.g., if conditions) to control the flow of your workflow.

`cancelled()` functions

Checks if the current job or step was canceled due to a manual intervention or some other reason.

`always()`:

function is used to make a step or job always run, regardless of the status of previous steps or jobs.

These status check functions and expressions can be used in conjunction with other workflow configuration options to create complex and conditional workflows that adapt to the outcomes of previous steps or jobs in your GitHub Actions pipeline.



Project Status Meeting - 2

Project Status Meeting - 2

Priority	Task	Assigned	Status	Comments/Issue
0	Understand Requirement	Alice	Completed	n.a
1	Unit Testing	Alice	In Progress	Production Database is unresponsive and laggy at times
2	Code Coverage	Alice	In Progress	
3	Containerization	Alice	Completed	n.a
4	Kubernetes Dev Deployment	Alice	Not started	
5	Dev Integration Testing	Alice	Not started	
6	Manual Approval	Alice	Not started	
7	Kubernetes Prod Deployment	Alice	Not started	
8	Prod Integration Testing	Alice	Not started	

Project Status Meeting - 2

Priority	Task	Assigned	Status	Comments/Issue
0	Understand Requirement	Alice	Completed	n.a
1	Unit Testing	Alice	Completed	Production Database is unresponsive and laggy at times
2	Code Coverage	Alice	Completed	
3	Containerization	Alice	Completed	
4	Kubernetes Dev Deployment	Alice	Not started	
5	Dev Integration Testing	Alice	Not started	
6	Manual Approval	Alice	Not started	
7	Kubernetes Prod Deployment	Alice	Not started	
8	Prod Integration Testing	Alice	Not started	



Job and Service Containers

Job Containers

```
Terminal
name: My Awesome App
on: push
jobs:
  unit-testing:
    runs-on: ubuntu-latest
    steps:
```



Language/ Runtime

- Node.js 18.17.1
- Perl 5.34.0
- Python 3.10.12
- Ruby 3.0.2p107



Package Management

- Miniconda 23.5.2
- Npm 9.6.7
- NuGet 6.6.1.2
- Pip 22.0.2



Tools

- Docker Client 24.0.5
- Terraform 1.5.6
- yamllint 1.32.0
- kubectl 1.28.1



CLI Tools

- Azure CLI 2.51.0
- AWS CLI 2.13.15
- GitHub CLI 2.33.0
- Vercel CLI 32.1.0



Browsers and Drivers

- Google Chrome
- Microsoft Edge
- Mozilla Firefox
- GeckoDriver



Cached Docker Image

- alpine:3.16
- debian:10
- node:18-alpine
- ubuntu:22.04

When creating a workflow, we are given the choice to select a runner for each job. Often, we opt for GitHub's hosted virtual machines to execute our workflows. These virtual machines are equipped with a preconfigured environment that includes various tools, packages, and settings designed for GitHub Actions.

For instance, the "ubuntu-latest" operating system on these virtual machines comes with preinstalled software, containing language runtimes, package management tools, command-line utilities, web browsers, cached Docker images, and much more.

Within a job, we can directly leverage these preinstalled software components to carry out our steps. However, there are

instances when we need a different version of a runtime or a package that isn't readily available in the preinstalled environment.

In such cases, we can compose additional steps in our workflow to install the required runtime versions and packages.

Job Containers

```
name: My Awesome App
on: push
jobs:
  unit-testing:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
      - name: Install NodeJS Version 20
      - name: Install Dependencies
      - name: Install Testing Packages
      - name: Run Tests
    run: npm test
```



Github Hosted Runner

Azure

OS - Ubuntu 22.04

1. Checkout Code - Time Taken 2 secs
2. Install NodeJS Version 20 - Time Taken 10 mins
3. Install Dependencies - Time Taken 3 mins
4. Install Testing Packages - Time Taken 45 mins
5. Run Tests - Time Taken 5 mins

In the given example, we are installing Node.js runtime version 20 and additional testing packages essential for our unit testing process.

When we trigger the workflow, it will execute all the defined steps using a GitHub hosted runner running the Ubuntu operating system.

It's worth noting that in certain situations, the time it takes to install specific runtimes and packages can significantly

increase the GitHub Actions billing costs. To mitigate this, one effective strategy is to employ job containers.

Job Containers

```
Terminal
name: My Awesome App
on: push
jobs:
  unit-testing:
    runs-on: ubuntu-latest

  steps:
    - name: Checkout Code
    - name: Install NodeJS Version 20
    - name: Install Dependencies
    - name: Install Testing Packages
    - name: Run Tests
      run: npm test
```

© Copyright KodeKloud



Github Hosted Runner



OS - Ubuntu 22.04

Docker Container - `node-and-packages:20`

1. Checkout Code - Time Taken 2 secs
2. Install Dependencies - Time Taken 3 mins
3. Run Tests - Time Taken 5 mins

A job container in GitHub Actions is a Docker container that is used to run the steps in a job. Job containers provide a number of benefits, including:

1. Isolation: Each job runs in its own container, which isolates it from other jobs and the host machine. This helps to prevent conflicts and security breaches.
2. Reproducibility: The same job container can be used to run the steps in a job on any machine, which ensures that the

results of the job are reproducible.

3. Security: Job containers can be configured to use a specific set of permissions, which helps to protect the host machine from malicious code.

To utilize a job container in GitHub Actions, you must specify the Docker image to be used within the job by configuring the "container" tag in your workflow file. Additionally, you have the flexibility to define other configurations, including credentials, environment variables, and volumes for mounting.

For instance, this workflow employs a job container to execute a custom container that includes Node.js runtime version 20 and the necessary testing packages. By utilizing this container, the need to install the Node.js runtime and test packages steps separately is eliminated.

During the execution of the workflow, GitHub creates a hosted runner with the Ubuntu operating system. Within this OS, it creates and runs the Docker container, executing all the defined steps within that container.

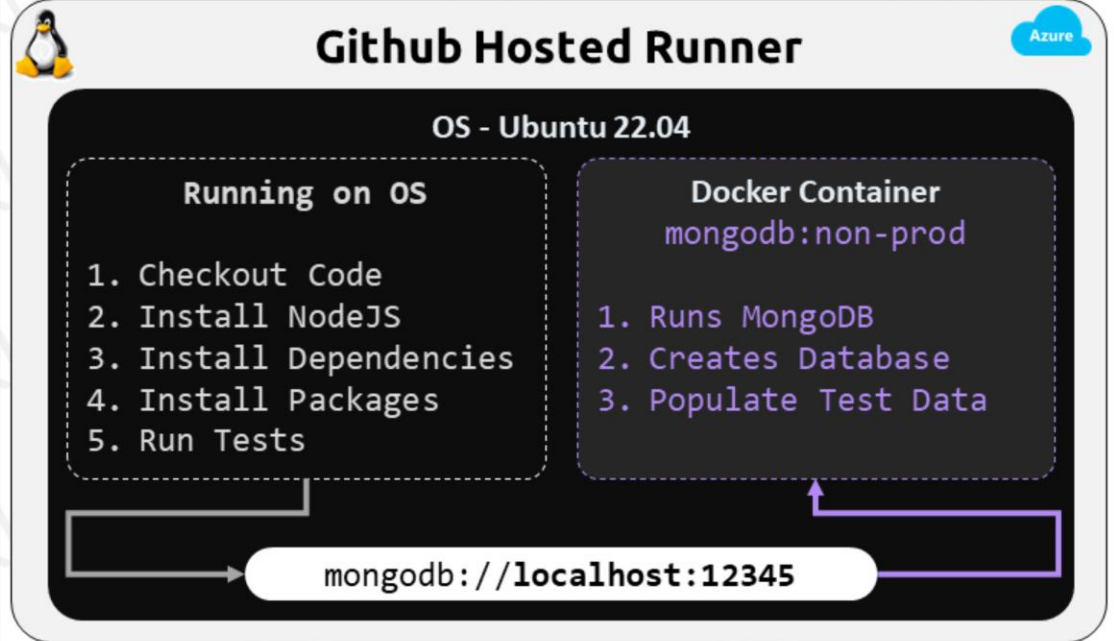
It's important to note that in this particular workflow, unit tests are executed while connected to the production database. This practice is not advisable, as it can potentially impact the production system's performance. For tasks like unit testing or code coverage analysis, it is recommended to use dedicated testing databases or services.

GitHub Actions offers a solution to address this issue.

Service Containers

```
name: My Awesome App
on: push
jobs:
  unit-testing:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Code
      - name: Install NodeJS Version 20
      - name: Install Dependencies
      - name: Install Testing Packages
      run: npm test
```



Much like Job containers, Service containers within GitHub Actions are Docker containers employed to host services essential for testing or operating your application within your workflow. For instance, your workflow may require executing unit tests that depend on a database and memory cache. You can set up service containers for each job within your workflow.

To utilize a service container, you must specify the Docker image by utilizing the "services" tag in your workflow file.

Additionally, you can configure various settings, such as environment variables to be set and the ports to expose.

In the provided example, the GitHub-hosted runner "ubuntu-latest" serves as the Docker host.

We have configured a service container to run a non-production MongoDB instance, labeled as "mongodb-service." To ensure proper connectivity, we've mapped port 20717 on the MongoDB service container to port 12345 on the Docker host. This port mapping is necessary because the job operates directly on the runner machine's operating system.

As a result, any step defined within this job will have access to the service container via the localhost and port 12345.

Service Container with Job Container

```
Terminal
name: My Awesome App
on: push
jobs:
  unit-testing:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Code
      - name: Install Dependencies
      - name: Run Tests
        run: npm test
```



Github Hosted Runner

Azure

OS - Ubuntu 22.04

Docker Container

node-and-packages:20

1. Checkout Code
2. Install Dependencies
3. Run Tests

Docker Container

mongodb:non-prod

1. Runs MongoDB
2. Creates Database
3. Populate Test Data

mongodb://mongodb-service:27017

Configuring jobs to run in a container simplifies networking configurations between the job and the service containers.

Docker containers on the same user-defined bridge network expose all ports to each other, so you don't need to map any of the service container ports to the Docker host. You can access the service container from the job container using the label you configure in the workflow.

This workflow configures a job that runs in a customized nodejs container and uses the ubuntu-latest GitHub-hosted runner as the Docker host for the container.

The workflow configures a service container with the label mongodb-service. All services must run in a container, so we use a custom mongodb container image.

When The workflow is executed it performs the following steps:

- 1.Checks out the repository on the runner
- 2.Installs dependencies
- 3.Runs unit tests by connected to the mongodb service container.

The hostname of the mongodb service is the label we configured in the workflow, in this case, mongodb-service. Because job and service Docker containers on the same user-defined bridge network open all ports by default, you'll be able to access the service container on the default mongodb port 27017.

Now that we understood Job and Service containers, let's implement them.

=====

Configuring containerized jobs simplifies networking setups between jobs and service containers.

This workflow configuration specifies a job that runs within a customized Node.js container. It utilizes the "ubuntu-latest" GitHub-hosted runner as the Docker host for this container. Additionally, the workflow configures a service container labeled as "mongodb-service." with a custom MongoDB container image.

When the workflow is executed, it will run 2 containers,
One is a mongodb service container, which runs mongodb and populates test data,
and
Within the job container it carries out the following steps:

1. Checks out the repository on the runner.
2. Installs dependencies.
3. Executes unit tests by connecting to the MongoDB service container.

The hostname of the MongoDB service corresponds to the label configured in the workflow, which in this case is "mongodb-service." Given that job and service Docker containers run within the same user-defined bridge network, all ports are exposed to each other, you'll have seamless access to the service container via the default MongoDB port, which is 27017.

Now that we have a grasp of Job and Service containers, let's proceed with their implementation.



Project Status Meeting - 3

Project Status Meeting - 3

Priority	Task	Assigned	Status	Comments/Issue
0	Understand Requirement	Alice	Completed	n.a
1	Unit Testing	Alice	Completed	n.a
2	Code Coverage	Alice	Completed	
3	Containerization	Alice	Completed	
4	Kubernetes Dev Deployment	Alice	In Progress	
5	Dev Integration Testing	Alice	In Progress	
6	Manual Approval	Alice	Not started	
7	Kubernetes Prod Deployment	Alice	Not started	
8	Prod Integration Testing	Alice	Not started	



KodeKloud

© Copyright KodeKloud

Follow us on <https://kodekloud.com/> to learn more about us.