

Python unittest

Unit Testing Basics

Unit testing involves writing test cases for each unit of code, which are small, isolated tests designed to validate the functionality of a specific part of the codebase. These tests are executed automatically and can be run repeatedly to make sure that changes to the code do not introduce new bugs.

Unit tests are typically written by the developers who write the corresponding code. They are executed frequently during the development process, especially before committing changes to a version control system, merging branches, or releasing new versions of the software.

The Role of Unit Testing in Software Development

Unit testing plays a critical role in the software development lifecycle by providing several key benefits:

Early Bug Detection: Unit tests help identify issues at an early stage, often immediately after code is written. This makes it easier and cheaper to fix bugs compared to

finding them later in the development process or after the software is deployed.

Improved Code Quality: Writing unit tests forces developers to think about edge cases, error handling, and the overall design of their code. This often leads to more thoughtful and strong code, as the process of writing tests can highlight potential weaknesses or areas for improvement.

Facilitates Refactoring: With a comprehensive suite of unit tests in place, developers can refactor or modify code with confidence. If a test fails after a change, it indicates that the change introduced a bug, allowing developers to quickly identify and address the issue.

Documentation: Unit tests can serve as additional documentation for the code. They provide concrete examples of how the code is intended to be used and what its expected behaviour is under various conditions. This can be particularly useful for new developers joining a project or for maintaining the code over time.

Supports Continuous Integration: Unit testing is a fundamental part of continuous integration (CI) practices. CI involves automatically building and testing code changes as they are integrated into a shared repository. By running unit tests as part of the CI

process, teams can make sure that new changes do not break existing functionality.

Key Concepts in Unit Testing

Understanding some key concepts in unit testing can help you write more effective tests:

Test Case: A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. Each test case should test one aspect of the code in isolation.

Test Suite: A test suite is a collection of test cases. It groups together tests that are related or that need to be run together.

Test Runner: A test runner is a component or tool that executes test cases and provides the results. It can be part of a testing framework like unittest, which we'll explore in this article.

Assertions: Assertions are statements used in test cases to check if the code behaves as expected. If an assertion fails, the test runner reports a failure. Common assertions include checking for equality, checking if a condition is true, or verifying that an exception is raised.

Fixtures: Fixtures are used to set up the necessary conditions for a test case to run. They can include

setting up databases, creating objects, or configuring the environment. In unittest, fixtures are often set up using the setup method and cleaned up using the teardown method.

Benefits of Unit Testing

The benefits of unit testing extend beyond just finding bugs. Here are some additional advantages:

Simplifies Integration: By ensuring that each unit works correctly in isolation, unit testing simplifies the process of integrating different parts of the codebase. When units are known to be correct, developers can focus on how they interact with each other.

Enhances Code Reusability: Unit tests can encourage developers to write modular, reusable code. When code is written with testing in mind, it is often designed to be more modular and decoupled, making it easier to reuse in different parts of the application.

Reduces Technical Debt: Consistently writing unit tests helps maintain code quality over time, reducing technical debt. Technical debt refers to the extra work required to fix issues that arise from poor code quality. By catching issues early, unit tests help prevent the accumulation of technical debt.

Increases Developer Confidence: Knowing that the code is thoroughly tested gives developers confidence in their work. This can lead to faster development cycles, as developers can focus on adding new features or improving existing ones without worrying about introducing new bugs.

Automated Testing: Unit tests can be run automatically, allowing for continuous testing without manual intervention. This automation is particularly valuable in agile development environments where frequent changes and deployments are common.

Creating a Basic Test Case:

```
import unittest

def add(a, b):
    return a + b

class TestMathOperations(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(1, 2), 3)
        self.assertEqual(add(-1, 1), 0)
        self.assertEqual(add(-1, -1), -2)

if __name__ == '__main__':
    unittest.main()
```

Arrange, Act, Assert (AAA): Follow the AAA pattern to structure your tests.

Arrange: Set up the necessary conditions and inputs for the test.

Act: Execute the code being tested.

Assert: Verify that the result is as expected.

```
class TestStringMethods(unittest.TestCase):  
  
    def setUp(self):  
        self.string = "hello world"  
  
    def test_upper(self):  
        # Arrange  
        expected = "HELLO WORLD"  
  
        # Act  
        result = self.string.upper()  
  
        # Assert  
        self.assertEqual(result, expected)
```

Running Unit Tests:

```
python test_script.py
```

```
python -m unittest discover
```

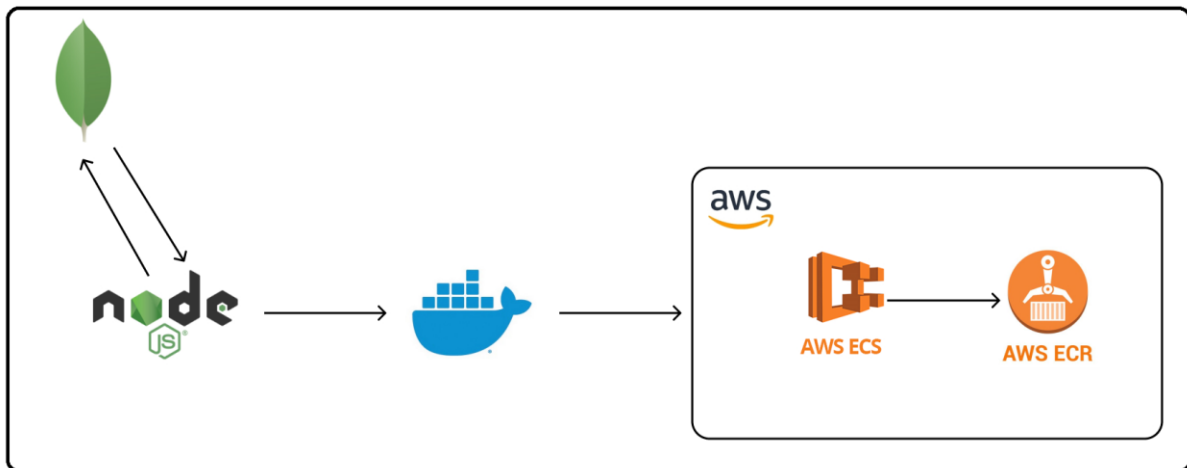
Build Docker image in pipeline:

Dockerfile: Defines how to build your application.

CI/CD Pipeline: Automates build, test, and deploy stages.

Secrets: Store Docker Hub credentials securely.

Deployment: Deploy the Docker image to a container orchestrator like Kubernetes.

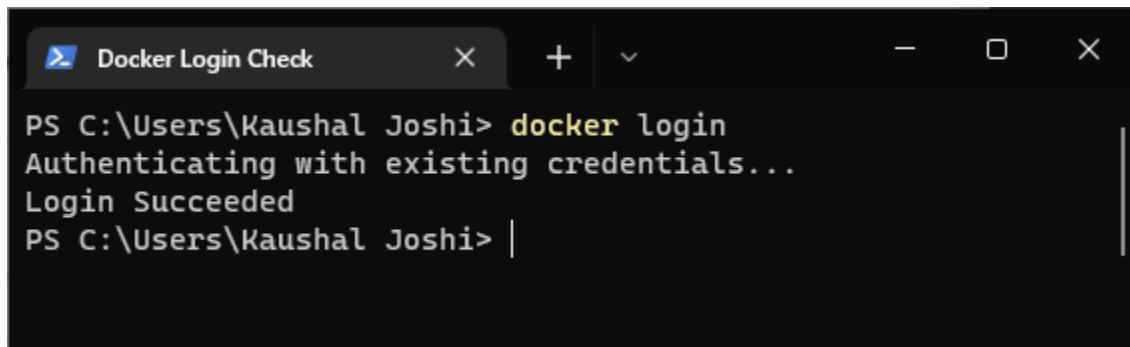


Docker Setup:

The only thing you need to do if you're using Windows or Mac is install the Docker desktop application. It installs everything you need and gives a nice GUI for interaction.

```
Docker Version Check x + v - □ x
PS C:\Users\Kaushal Joshi> docker --version
Docker version 20.10.14, build a224086
PS C:\Users\Kaushal Joshi> |
```

Docker Login:



```
PS C:\Users\Kaushal Joshi> docker login
Authenticating with existing credentials...
Login Succeeded
PS C:\Users\Kaushal Joshi> |
```

How to Dockerize My Project:

Dockerize, I mean setting up your existing project with Docker and containerizing it.

Create a file named Dockerfile without any extension in the root of your project directory. It contains the code required to build a Docker image and run the Dockerized app as a container.

How to Configure the Dockerfile

As a bare minimum configuration, paste the following code in the **Dockerfile**.

Create a Docker Image:

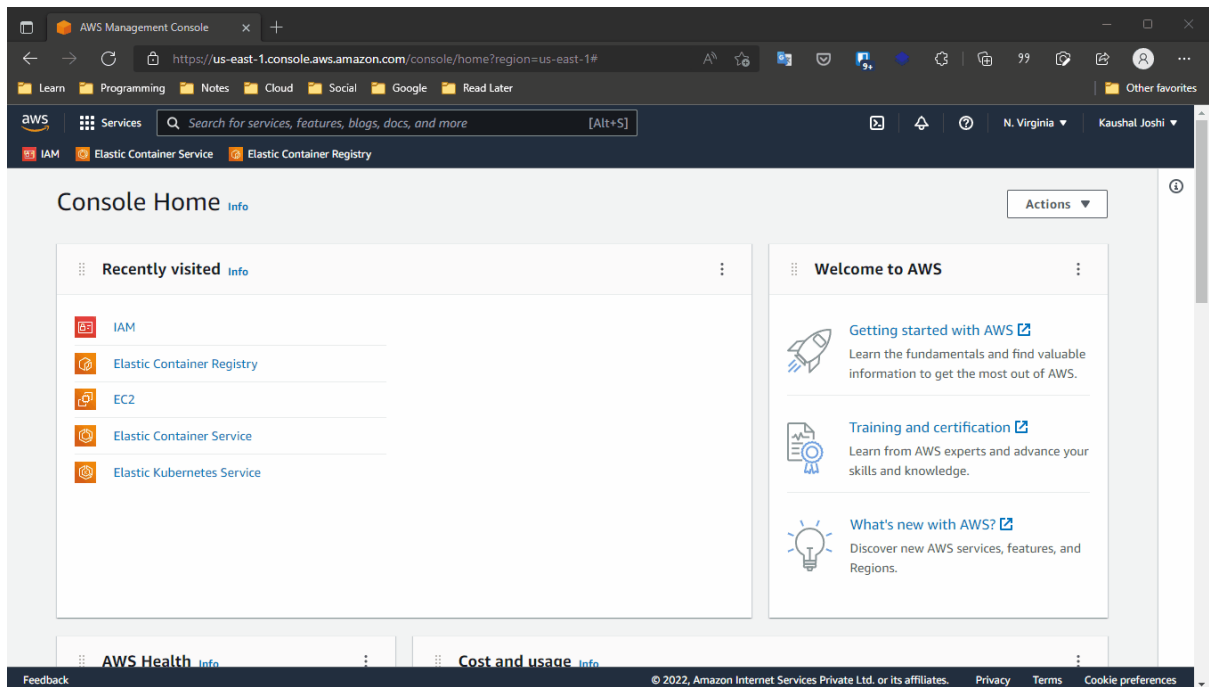
```
docker build -t <name-tag>
```

```
docker run -p 3000:3000 <name-tag>
```

How to Set Up AWS CLI

```
aws cli
```

How to Install AWS CLI



Access Key - Programmatic Access is checked when you enter the name of new user.

Add a user policy that gives full access of ECS. The name of the policy is AmazonECS_FullAccess.

Note down access key ID and Secret access key, as we'll have to use these later.

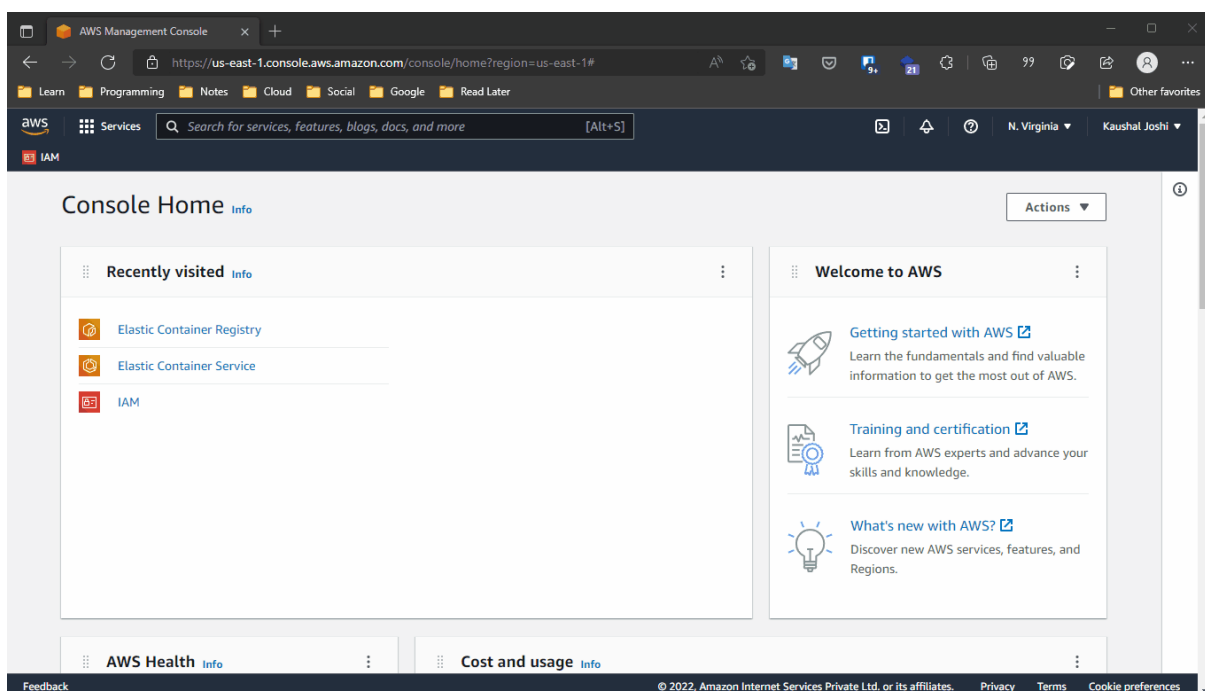
aws configure

```
PS C:\Users\Kaushal Joshi> aws configure list
      Name                               Value                               Type    Location
      ----                               -
      profile                            <not set>                          None     None
      access_key                         *****OIMM                        shared-credentials-file
      secret_key                         *****nJx5                        shared-credentials-file
      region                             us-east-1                          config-file  ~/.aws/config
PS C:\Users\Kaushal Joshi> |
```

How to Create a Repo in ECR

```
aws ecr create-repository --repository-name  
<repo_name> --region <region_name>
```

```
Creating a repo in ECR x + v
PS C:\Users\Kaushal Joshi> aws ecr create-repository --repository-name demo-repo --region us-east-1
{
  "repository": {
    "repositoryArn": "arn:aws:ecr:us-east-1:594208535949:repository/demo-repo",
    "registryId": "594208535949",
    "repositoryName": "demo-repo",
    "repositoryUri": "594208535949.dkr.ecr.us-east-1.amazonaws.com/demo-repo",
    "createdAt": "2022-04-20T16:19:24+05:30",
    "imageTagMutability": "MUTABLE",
    "imageScanningConfiguration": {
      "scanOnPush": false
    },
    "encryptionConfiguration": {
      "encryptionType": "AES256"
    }
  }
}
```



How to Push a Docker Image to ECR

```
aws ecr get-login-password --region <region_name>
```

```
aws ecr --region <region> | docker login -u AWS -p <encrypted_token> <repo_uri>
```

-u AWS: Default user provided by AWS.

-p <encrypted_token>: Password we retrieved in the last step.

repo_uri: URI of our repository.

How to Tag a Local Docker Image

```
docker tag <source_image_tag> <target_ecr_repo_uri>
```

How to Push the Docker Image to ECR

```
docker push <ecr-repo-uri>
```

Amazon ECR > Repositories > members-only > sha256:b4f1d2931472339c8967bb4a99872fdcf3b93f6f6b639e099ad9e7007e02d14b

Image details

Scan

Image URI 594208535949.dkr.ecr.us-east-1.amazonaws.com/members-only:latest	
Digest sha256:b4f1d2931472339c8967bb4a99872fdcf3b93f6f6b639e099ad9e7007e02d14b	
Image tags latest	Repository members-only
Pushed at April 20, 2022, 13:09:43 (UTC+05.5)	Size (MB) 362.27
Type	
Image manifest type application/vnd.docker.distribution.manifest.v2+json	Artifact media type application/vnd.docker.container.image.v1+json
Basic scanning	
Scan status Complete, April 20, 2022, 17:41:26 (UTC+05.5) The scan was completed successfully.	
Vulnerabilities 6 Critical + 1369 others (details)	