Topics

- Software testing
- Continuous integration
- Continuous deployment
- Containerisation
- Infrastructure

Introduction to Software Testing

Software testing is the systematic process of evaluating a software application to ensure it meets the specified requirements and is free of defects. It plays a crucial role in delivering a high-quality product that provides value to users and stands the test of time.

Software Testing Methods

Software testing can be broadly categorized into two methods: manual testing and automated testing.

Manual Testing

This is the traditional approach where testers manually execute test cases without the assistance of tools or scripts. It involves checking all the features of an application by hand to ensure they behave as expected.

Pros:
- Allows for human intuition and creativity.
- Useful for exploratory, usability, and ad-hoc testing.

Cons:
  - Time-consuming and labor-intensive.
  - Prone to human error.
  - Not scalable for large, complex systems.

Automated Testing

Automated testing uses software tools to execute pre-scripted tests on the application automatically. This method is ideal for repetitive tasks and regression testing, where the same tests need to be run multiple times.

Pros:
  - Faster execution of tests.
  - More reliable and consistent results.
  - Saves time in the long run.

Cons:
  - Requires initial investment in tools and scripting.
  - Less effective for tests requiring human judgment.

Automated Software Testing Levels

Automated testing is organized into different levels, each focusing on specific aspects of the software.

Unit Testing

Tests individual components or units of code (e.g., functions, methods) in isolation.
The purpose of unit testing is to ensure that each unit performs as designed.

Integration Testing

Tests the interaction between integrated units or components.
The purpose is to detect issues in the interfaces and interactions between modules.

≠System Testing

Tests the complete, integrated system as a whole.
System testing is meant to validate end-to-end system specifications.

Acceptance Testing

Conducted to determine if the system meets the acceptance criteria and is ready for deployment.
- User Acceptance Testing (UAT): Real users test the system in real-world scenarios.

Why We Need Software Testing

1. Maintain Product Quality

Regular testing ensures that new changes do not break existing functionality.

2. Facilitates Changes and Updates

A well tested system allows developers to make changes confidently.

3. Increases Customer Satisfaction

A well-tested product meets user expectations. Satisfied customers are more likely to continue using and recommending the product.

4. Enhances Security

Testing is crucial for identifying potential security flaws before they can be exploited.

5. Cost-Effectiveness

Fixing issues early reduces the cost and effort required to fix them later in production.

6. Promotes Team Collaboration

Software testing whether manual or automated gives results that provide feedback for developers.

Types of Software Testing

Functional Testing

Tests the functionality of the system
- Examples: Unit tests, integration tests, system tests.

Non-Functional Testing

Examples:
- Performance Testing: Checks responsiveness and stability under load.
- Security Testing: Identifies vulnerabilities.
- Usability Testing: Assesses user-friendliness.

Regression Testing

Re-running functional and non-functional tests to ensure that previously developed software still performs after new changes are added to the software.

Smoke Testing

A quick set of tests to check the basic functionality. Smoke testing can be used to quickly verify that the critical functionalities are working.

Test Automation Tools and Frameworks

Unit Testing Frameworks

- pytest: For Python applications.
- JUnit: For Java applications.
- NUnit: For .NET applications.

UI Testing Tools

- Selenium WebDriver: Automates web browsers.
- Appium: Automates mobile applications.

Continuous Integration Tools

- Jenkins: Open-source automation server.
- Travis CI: Hosted continuous integration service.
- CircleCI: Continuous integration and delivery platform.
- Github actions

Best Practices in Software Testing

- Start Early
- Test Often
- Prioritize Testing
- Automate Where Possible
- Monitor and Measure
- Ensure comprehensive coverage

Let’s write simple unit test for our Student Model

In **student/models.py** we will add a method that returns a students full name and then write a test to ensure that it always functions as expected.

```
def full_name(self):
  return f"{self.first_name}"
```

We will then add a test that will always ensure that this method always functions as expected.

Django requires you put your tests in a file that contains the word test.
student/tests.py

```
import datetime

from django.test import TestCase

from .models import Student


class StudentModelTests(TestCase):
   def setUp(self):
       self.student = Student(
           first_name="James",
           last_name="Mwai",
           date_of_birth=datetime.date(2000, 9, 17),
           gender="male",
           registration_number="123456",
           email="james@gmail.com",
           phone_number="254123456",
       )

   def test_full_name_contains_first_name(self):
       self.assertIn(self.student.first_name, self.student.full_name())

   def test_full_name_contains_last_name(self):
    self.assertIn(self.student.last_name, self.student.full_name())
```

To run the above tests,

```
python manage.py test student
```

On of the above tests will fail. Fix the bug in the code and ensure that both tests are now passing.

More testing examples

Testing a Form

Forms are critical for validating user input. In our project we have StudentRegistrationForm that accepts the first name, last name, email and other attributes for creating a student.

Our StudentRegistrationForm:

student/forms.py

```
from django import forms
from .models import Student

class StudentRegistrationForm(forms.ModelForm):
    class Meta:
        model = Student
        fields = '__all__'
```

Test Case for StudentForm:

```
from django.test import TestCase
from .forms import StudentForm

class StudentFormTest(TestCase):
    def test_student_form_valid(self):
        form_data = {
                    'first_name': Glory,
                    'last_name': Maina,
                    'email': 'gmaina@example.com'
                            } #Add all the other required attributes

        form = StudentForm(data=form_data)
        self.assertTrue(form.is_valid())
```

```
def test_student_form_invalid(self):
    form_data = {'first_name': Glory, 'last_name': Maina}
    form = StudentForm(data=form_data)
    self.assertFalse(form.is_valid()) # email and others are required
    self.assertIn('email', form.errors)
```

5. Testing an API View (Django Rest Framework)

In our simple project we have created an API endpoint for listing students using Django Rest Framework's APIView.

Our Project's Student List API View:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from student.models import Student
from .serializers import StudentSerializer

class StudentListView(APIView):
    def get(self, request):
        students = Student.objects.all()
        serializer = StudentSerializer(students, many=True)
        return Response(serializer.data)
```

Implement the test case for this api endpoint in your project as follows.

Test Case for StudentListAPIView:

You can put this code in api/tests.py

```
from django.urls import reverse
from rest_framework.test import APITestCase
from rest_framework import status
from student.models import Student
from .serializers import StudentSerializer

class StudentListAPIViewTest(APITestCase):
    def setUp(self):
        self.student = Student.objects.create(
            first_name="Glory",
            last_name="Maina",
            date_of_birth="2005-05-15",
            email="gmaina@gmail.com",
            course=self.course)

    def test_get_student_list(self):
        url = reverse('student_list')
        response = self.client.get(url)
        students = Student.objects.all()
        serializer = StudentSerializer(students, many=True)
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(response.data, serializer.data)
```

Continuous Integration, Deployment and Delivery.

Continuous Integration

CI is where members of a software development team integrate their work frequently. Each integration is verified by an automated build system which should run tests and code analysis to ensure the incoming code is correct and does not break any existing functionality.

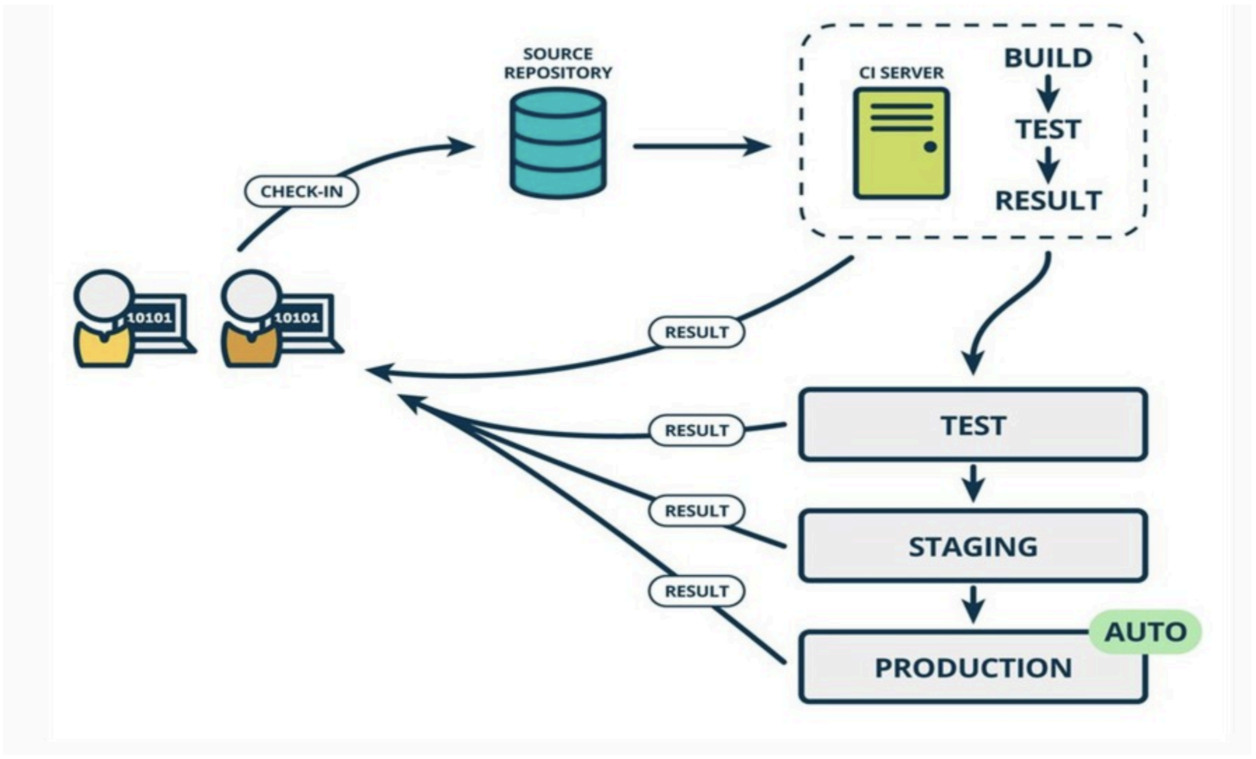
For Developers to be able to integrate their work they need to;

- Use source control
- Write tests
- Use an automated build system responsible for checking and merging their work.
- Run tests after every integration

The first thing is to ensure we are pushing our code to a source control, in our case github.

Continuous Deployment.

Continuous deployment builds on top of continuous integration to allow continuous deployment of new software features as they are being added.



In a typical web development environment, you can have three environment

- Dev
- Staging/QA
- Production

You continuously deploy to dev as tests pass. Once you want to deploy to production, you can first deploy to a staging/qa environment to do user acceptance testing to ensure everything is working properly. If all is well you can then push to production.

Most startups use what is known as the Git-Flow collaboration method where developers check-in their code to develop branch through pull requests submitted from feature branches. Any changes merged to develop branch are instantly deployed to the dev server. To deploy to staging you can merge all dev changes to master and master should instantly deploy to staging server.

To deploy to production, you typically create a release branch out of master which is tagged and deployed to the production server.

Setting Up GitHub Actions for Continuous Integration

1. **Navigate to Your GitHub Repository**:
    - Open your Django project on GitHub.
    - Go to the **Actions** tab.
2. **Choose a Workflow**:
    - GitHub Actions will suggest workflows; select **Python application**.
3. **Configure Workflow for Django**:
    - GitHub will create a YAML file (`django-test.yml`) in `.github/workflows/`.
    - Edit it as follows to install Django, set up your database, and run tests:

```yaml
name: Django CI

on:
  pull_request:
    branches:
      - dev

jobs:
  test:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:13
        env:
          POSTGRES_DB: test_db
          POSTGRES_USER: test_user
          POSTGRES_PASSWORD: test_password
        ports:
          - 5432:5432
        options: >-
          --health-cmd="pg_isready -U test_user"
          --health-interval=10s
          --health-timeout=5s
          --health-retries=5

    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v3
        with:
          python-version: '3.9'

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Run migrations and tests
        env:
          DATABASE_URL:
postgres://test_user:test_password@localhost:5432/test_db
        run: |
          python manage.py migrate
          python manage.py test
```

4. **Commit the Workflow**:
   - Commit and push this YAML file to your repository.
   - GitHub will trigger the workflow automatically on pushes to the main branch or pull requests.
5. **Check CI Status**:
   - Go to the **Actions** tab to see your workflow running.
   - If successful, it will show a green checkmark. If it fails, debug based on error messages.

Set Up Github Actions for Continuous Deployment

Assuming you want to deploy your app on **Heroku**:

1. **Set Up Heroku**:
   - Create an account on Heroku.
   - Install the Heroku CLI on your computer.
   - Create a new app on Heroku and note the app name.
2. **Add Heroku API Key to GitHub Secrets**:
   - In your GitHub repo, go to **Settings > Secrets > Actions**.

- Add `HEROKU_API_KEY` as a secret, which you can find by running `heroku auth:token` in your terminal.

3. **Create a GitHub Action for CD**:
    - Add another job to your workflow file or create a new YAML file for deployment.

```
name: Django CI/CD

on:
  push:
    branches:
      - dev

jobs:
  test:
    # CI job as above
  deploy:
    needs: test  # Run only if tests pass
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Login to Heroku
        env:
          HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
        run: |
          echo $HEROKU_API_KEY | docker login --username=_ --password-stdin
registry.heroku.com

      - name: Deploy to Heroku
        run: |
          heroku container:push web --app YOUR_HEROKU_APP_NAME
          heroku container:release web --app YOUR_HEROKU_APP_NAME
```

4. **Commit and Push**:
    - Pushing these changes triggers the CI/CD pipeline.
    - Once CI tests pass, the CD job will deploy the app to Heroku.