Modulo 8: Fundamentos de Integración Continua

Clase 2: Integración Continua con GitHub Actions

Crear un Pipeline en GitHub Actions para un Proyecto Python/Django

Parte 1: Preparar tu proyecto

1. Crea una carpeta para tu proyecto y entra en ella:

```
mkdir demo-pipeline
cd demo-pipeline
```

1. Crea un entorno virtual y activalo:

```
python -m venv venv
# en windows
venv\Scripts\activate
# en osx/linux
source venv/bin/activate
```

1. Instala las dependencias necesarias (django y pytest):

```
pip install django
```

1. Crea un proyecto Django:

```
django-admin startproject myproject .
```

1. Crea una app en tu proyecto y agregala a tu proyecto en el archivo `settings.py`:

```
python manage.py startapp myapp
```

1. Momento de hacer el primer commit:

Ejemplo de archivo `.gitignore`:

```
# .gitignore
venv/
*.pyc
__pycache__/
```

```
git init
# create gitignore for python
git add .
git commit -m "Initial commit"
```

1. Agrega una vista simple en `myapp/views.py`:

```
from django.http import JsonResponse

def ping(request):
    return JsonResponse({'ping': 'pong'})
```

1. Agrega una URL para tu vista en `myproject/urls.py`:

```
from django.contrib import admin
from django.urls import path
from myapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('ping/', views.ping, name='ping'),
]
```

1. Crea un archivo `requirements.txt`:

```
pip freeze > requirements.txt
```

1. Momento de hacer el segundo commit:

```
git add .
git commit -m "Add ping view"
```

Parte 2: Crear pruebas automatizadas

1. Crea un archivo `tests.py` en `myapp/tests.py`:

```
from django.test import TestCase, Client
from myapp.views import sumar

# Test del endpoint ping
class PingTests(TestCase):
    def setUp(self):
        self.client = Client()

    def test_endpoint_ping_responde_ping_pong(self):
        response = self.client.get('/ping/')
        self.assertEqual(response.status_code, 200)
        self.assertJSONEqual(response.content, {'ping': 'pong'})
```

1. Ejecuta las pruebas localmente:

```
python manage.py test
```

1. Momento de hacer el tercer commit:

```
git add .
git commit -m "Add tests"
```

Parte 3: Subir tu proyecto a GitHub

1. Crea un repositorio en GitHub.
2. Agrega el repositorio remoto a tu proyecto:

```
git remote add origin https://github.com/tu-usuario/demo-pipeline.git
```

1. Sube tus cambios a GitHub:

```
git push -u origin main
```

Parte 4: Crear un archivo de workflow de GitHub Actions

1. Crea un archivo `.github/workflows/ci.yml`:

```
name: Django CI

on:
  pull_request:
    branches: [main]
  push:

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: 3.11
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Run tests
        run: |
          python manage.py test
```

1. Commit y push tus cambios:

```
git add .
git commit -m "Add GitHub Actions workflow"
git push
```

Parte 5: Verificar el pipeline en GitHub

1. Ve a tu repositorio en GitHub.
2. Navega a la pestaña `Actions`.
3. Verifica que el pipeline se haya ejecutado correctamente.
4. Crea una nueva rama:

```
git checkout -b feature/include-response-date
```

1. Agrega a la vista de ping que incluya la fecha actual:

```
from django.http import JsonResponse
from datetime import datetime

def ping(request):
    return JsonResponse({'ping': 'pong', 'date': datetime.now().isoformat()
```

1. Al hacer este cambio, el sobre nuestra vista fallará, podemos verlo al ejecutar los tests localmente:

```
python manage.py test
```

1. Commit y push tus cambios:

```
git add .
git commit -m "Agrega la fecha actual a la vista de ping"
git push origin feature/include-response-date
```

1. Ve a GitHub y crea un Pull Request hacia main
2. Ver en la pestaña Actions que el test falla.
3. Corrige el test, haz commit y push nuevamente.

```
from datetime import datetime
from unittest.mock import patch

from django.test import TestCase


class PingTestCase(TestCase):
    def test_ping(self):
        fecha_esperada = "2024-03-20T12:00:00"
        with patch("myapp.views.datetime") as mock_datetime:
            mock_datetime.now.return_value = datetime.fromisoformat(fecha_e
            response = self.client.get("/ping/")
            self.assertEqual(response.status_code, 200)
            data = response.json()
            self.assertEqual(data["ping"], "pong")
            self.assertIsInstance(data["date"], str)
            self.assertEqual(data["date"], fecha_esperada)
```

1. Commit y push tus cambios:

```
git add .
git commit -m "Corrige el test"
git push origin feature/include-response-date
```

1. Verifica que ahora pase y el PR quede listo para aprobarse

Trabajo Grupal Bloque A

Modifica el pipeline con prueba fallida y verifica bloqueo de Pull Request

Verificar que un Pull Request no se pueda aprobar mientras exista una prueba que falla en GitHub Actions. 1. Cree una regla para proteger la rama main, se deben cumplir las siguientes condiciones: - Requiere un PR - Requiere verificación de estado de las pruebas 2. Cree un cambio que rompa las pruebas, verifique localmente y haga el commit y push a la rama. 3. Abrir un Pull Request hacia la rama principal (main). 4. Observe que: - La pestaña Actions: La indicación "Some checks were not successful" en el PR. - GitHub bloquea el merge si se exige pasar las pruebas. 5. Corregir el test para que pase, hacer commit y push nuevamente. 6. Verificar que el PR ahora se puede aprobar.

Trabajo Grupal Bloque B

Cambia el trigger del pipeline y verifica ejecución sólo en push

1. Modificar el archivo ci.yml para que el pipeline se ejecute solo en push a una rama creada en base a “main”, y no en PR.
2. Instrucciones.
    - Editar el archivo .github/workflows/ci.yml.
    - Hacer commit y push del cambio a la rama nueva.
    - Abrir un PR (No se ejecutará ningún pipeline).
3. Verificar en la pestaña Actions que el pipeline se ejecutó con el push a la rama nueva.