Sesión 7: Automatización de pruebas con Github Actions

Duración: 45 minutos

🎯 Objetivos de aprendizaje

Al finalizar esta sesión, el participante será capaz de:

- Configurar pipelines de CI/CD con GitHub Actions para testing
- Implementar estrategias de testing automatizado por tipo y entorno
- Configurar matrix testing para múltiples versiones de Python
- Integrar code coverage, security scanning y quality gates
- Crear workflows eficientes con caching y paralelización
- Implementar deployment condicional basado en testing

📋 Contenido

1. CI/CD y Testing - Visión General (8 min)

Analogía: CI/CD es como una cadena de producción automatizada. Cada commit es una pieza que pasa por estaciones de control de calidad (tests) antes de llegar al producto final (deployment).

¿Por qué automatizar testing?

```
🎯 PROBLEMAS SIN AUTOMATIZACIÓN:
├── Tests manuales → Lentos, inconsistentes
├── Regressions no detectadas → Bugs en producción
├── Feedback tardío → Fixes costosos
├── Integración riesgosa → "Works on my machine"
└── Deploy manual → Propenso a errores

✅ BENEFICIOS DE AUTOMATIZACIÓN:
├── Feedback inmediato → Fail fast
├── Consistency → Mismo entorno siempre
├── Paralelización → Tests simultáneos
├── Quality gates → No deploy si falla
└── Confidence → Deploy seguro
```

Filosofía de testing en CI/CD:

🔄 CONTINUOUS INTEGRATION FLOW:

Code Push → Build → Unit Tests → Integration Tests → Security Tests → Deploy

⚡ FAIL FAST PRINCIPLE:

```
├── Unit tests (2-5 min) → Feedback rápido
├── Integration tests (5-15 min) → Verificación completa
├── E2E tests (15-45 min) → Validación final
└── Security/Performance → Validación exhaustiva
```

2. GitHub Actions - Arquitectura y Conceptos (10 min)

Componentes principales:

```
📁 WORKFLOW (.github/workflows/*.yml)
|
├── 🎬 EVENT (push, PR, schedule, manual)
| |
| ├── 🏢 JOB (runs-on: ubuntu-latest)
| | |
| | ├── 📋 STEP (uses: actions/checkout@v3)
| | ├── 📋 STEP (run: pytest tests/)
| | └── 📋 STEP (uses: codecov/codecov-action@v3)
| |
| └── 🏢 JOB (runs-on: windows-latest)
|     └── 📋 STEP (run: pytest tests/)
```

Triggers más comunes:

yaml

```yaml
# Push a branches específicas
on:
  push:
    branches: [main, develop]
    paths: ['src/**', 'tests/**']

# Pull requests
on:
  pull_request:
    branches: [main]

# Schedule (cron)
on:
  schedule:
    - cron: '0 2 * * *'  # Daily at 2 AM

# Manual trigger
on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to deploy'
        required: true
        default: 'staging'
```

Runners y environments:

```
🖥️ GITHUB-HOSTED RUNNERS:
├── ubuntu-latest (más común para Python)
├── windows-latest
├── macos-latest
└── ubuntu-20.04, ubuntu-18.04 (versiones específicas)

🏠 SELF-HOSTED RUNNERS:
├── Control total del entorno
├── Acceso a recursos internos
├── Software específico pre-instalado
└── Mejor para workloads intensivos
```

3. Workflow de Testing Básico a Avanzado (10 min)

Nivel 1: Testing básico

```yaml
name: Basic Testing
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with:
          python-version: '3.9'
      - run: pip install -r requirements.txt
      - run: pytest tests/
```

Nivel 2: Matrix testing

```yaml
strategy:
  matrix:
    python-version: ['3.8', '3.9', '3.10', '3.11']
    os: [ubuntu-latest, windows-latest, macos-latest]
    include:
      - python-version: '3.12'
        os: ubuntu-latest
        experimental: true
  fail-fast: false  # Continuar aunque un job falle
```

Nivel 3: Testing por capas

```yaml
```

```yaml
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Lint code
        run: |
          flake8 src/
          black --check src/
          mypy src/

  unit-tests:
    needs: lint
    strategy:
      matrix:
        python-version: ['3.9', '3.10', '3.11']
    steps:
      - name: Unit tests
        run: pytest tests/unit/ --cov=src

  integration-tests:
    needs: unit-tests
    services:
      postgres:
        image: postgres:13
        env:
          POSTGRES_PASSWORD: postgres
    steps:
      - name: Integration tests
        run: pytest tests/integration/

  security-tests:
    needs: lint
    steps:
      - name: Security scanning
        run: |
          bandit -r src/
          safety check
```

4. Optimización de Performance (7 min)

Caching estratégico:

yaml

```yaml
- name: Cache Python dependencies
  uses: actions/cache@v3
  with:
    path: ~/.cache/pip
    key: ${{ runner.os }}-pip-${{ hashFiles('**/requirements.txt') }}
    restore-keys: |
      ${{ runner.os }}-pip-

- name: Cache test results
  uses: actions/cache@v3
  with:
    path: .pytest_cache
    key: pytest-${{ runner.os }}-${{ hashFiles('tests/**/*.py') }}
```

Paralelización inteligente:

```yaml
# Parallel test execution
- name: Run tests in parallel
  run: |
    pytest tests/unit/ -n auto --dist worksteal
    pytest tests/integration/ -n 2
    pytest tests/security/ --maxfail=1

# Job parallelization
jobs:
  test-unit:
    # Fast feedback
  test-integration:
    # Medium feedback
  test-e2e:
    # Slow but comprehensive
```

Conditional execution:

```yaml
```

```yaml
- name: Run expensive tests only on main branch
  if: github.ref == 'refs/heads/main'
  run: pytest tests/performance/ --slow

- name: Deploy only if all tests pass
  if: success() && github.ref == 'refs/heads/main'
  run: ./deploy.sh
```

Artifacts y reporting:

```yaml
- name: Upload test results
  uses: actions/upload-artifact@v3
  if: always()
  with:
    name: test-results-${{ matrix.python-version }}
    path: |
      junit.xml
      coverage.xml
      htmlcov/

- name: Upload to Codecov
  uses: codecov/codecov-action@v3
  with:
    file: ./coverage.xml
    flags: unittests
    name: codecov-umbrella
```

5. Integration con Servicios Externos (5 min)

Database services:

```yaml
```

```yaml
services:
  postgres:
    image: postgres:13
    env:
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: test_db
    options: >-
      --health-cmd pg_isready
      --health-interval 10s
      --health-timeout 5s
      --health-retries 5

  redis:
    image: redis:6
    options: >-
      --health-cmd "redis-cli ping"
      --health-interval 10s
      --health-timeout 5s
      --health-retries 5
```

External API testing:

```yaml
- name: Setup test environment
  run: |
    docker-compose -f docker-compose.test.yml up -d
    sleep 30  # Wait for services

- name: Run API tests
  run: |
    pytest tests/api/ --api-url=http://localhost:8080

- name: Cleanup
  if: always()
  run: docker-compose -f docker-compose.test.yml down
```

Secrets management:

```yaml
```

```yaml
- name: Test with production-like secrets
  env:
    DATABASE_URL: ${{ secrets.TEST_DATABASE_URL }}
    API_KEY: ${{ secrets.TEST_API_KEY }}
  run: pytest tests/integration/
```

6. Quality Gates y Deployment (5 min)

Coverage thresholds:

yaml

```yaml
- name: Coverage check
  run: |
    coverage run -m pytest tests/
    coverage report --fail-under=80
    coverage xml

- name: Quality gate
  uses: sonarqube-quality-gate-action@master
  env:
    SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

Deployment pipeline:

yaml

```yaml
deploy:
  needs: [test-unit, test-integration, security-scan]
  if: github.ref == 'refs/heads/main'
  environment:
    name: production
    url: https://myapp.example.com
  steps:
    - name: Deploy to production
      run: |
        echo "All tests passed ✅ "
        echo "Deploying to production..."
        ./scripts/deploy.sh production

    - name: Smoke test production
      run: |
        curl -f https://myapp.example.com/health
        pytest tests/smoke/ --base-url=https://myapp.example.com
```

Rollback strategy:

```yaml
- name: Rollback on failure
  if: failure()
  run: |
    echo "Deployment failed, rolling back..."
    ./scripts/rollback.sh

- name: Notify team
  uses: 8398a7/action-slack@v3
  if: always()
  with:
    status: ${{ job.status }}
    text: "Deployment ${{ job.status }} for commit ${{ github.sha }}"
```

⚡ Best Practices para GitHub Actions

Performance:

- ✅ Use caching para dependencies y test results
- ✅ Ejecuta tests en paralelo cuando sea posible

- ✅ Fail fast en lint/syntax errors
- ✅ Matrix testing para compatibilidad

Reliability:

- ✅ Pin action versions (v3, no @main)
- ✅ Set timeouts para evitar workflows colgados
- ✅ Use conditional steps para diferentes branches
- ✅ Implement proper error handling

Security:

- ✅ Use secrets para información sensible
- ✅ Restrict permissions (GITHUB_TOKEN)
- ✅ Validate external inputs
- ✅ Use trusted actions only

Maintainability:

- ✅ Organize workflows por propósito
- ✅ Use reusable workflows
- ✅ Document complex logic
- ✅ Monitor workflow performance

🔧 Estrategias por Tipo de Proyecto

API/Microservice:

```
Workflow optimizado:
├── Lint + Type check (2 min)
├── Unit tests (3 min)
├── Integration tests con DB (5 min)
├── API contract tests (3 min)
├── Security scan (2 min)
└── Deploy + smoke test (5 min)
Total: ~20 minutos
```

Web Application:

```
Workflow completo:
├── Frontend build + test (5 min)
├── Backend unit tests (4 min)
├── E2E tests con browser (15 min)
├── Performance tests (10 min)
├── Security + accessibility (5 min)
└── Deploy staging + production (10 min)
Total: ~49 minutos
```

Library/Package:

```
Workflow exhaustivo:
├── Multi-version matrix (Python 3.8-3.11)
├── Multi-OS testing (Linux, Windows, macOS)
├── Documentation build
├── Package build + test install
├── Publish to TestPyPI
└── Conditional publish to PyPI
```

📝 Resumen Final

1. **GitHub Actions** automatiza testing de forma confiable y escalable
2. **Matrix testing** asegura compatibilidad across versions y OS
3. **Caching y paralelización** optimizan tiempo de feedback
4. **Quality gates** previenen deployment de código defectuoso
5. **Services** permiten testing con dependencias reales
6. **Conditional workflows** adaptan testing por branch/context
7. **Monitoring y artifacts** proporcionan observabilidad completa

---

🏃 Actividad Práctica Sugerida

Ejercicio 1 - Pipeline básico:

1. Crear workflow que ejecute pytest en Python 3.9
2. Añadir coverage reporting con codecov
3. Configurar quality gate (>80% coverage)

Ejercicio 2 - Pipeline avanzado:

1. Implementar matrix testing (Python 3.8-3.11)
2. Separar unit/integration tests en jobs diferentes
3. Añadir security scanning con bandit
4. Configurar deployment condicional a staging

Tiempo estimado: 25-30 minutos adicionales