Todo-List Project Enhanced

Todo Application Testing Project

Project Overview

This final project combines various testing methodologies and DevOps practices to create a comprehensive testing suite for the Todo application. The project will demonstrate proficiency in different testing levels, automation, and deployment strategies.

1. Testing Layers

1.1 Unit Tests

- Test individual components and functions:
  - User authentication methods
  - Task CRUD operations
  - Password hashing functions
  - OAuth integration functions
  - Data models and relationships
  - Form validation
- Use pytest as the testing framework
- Implement test fixtures for database and authentication
- Achieve minimum 80% code coverage

1.2 Integration Tests

- Test interactions between components:
  - Database operations
  - OAuth provider integration
  - Session management
  - User authentication flow
  - Task management workflow
- Use pytest-integration for integration testing
- Implement mock OAuth providers for testing

1.3 End-to-End Tests (Acceptance)

- Use Selenium or Playwright for E2E testing
- Test complete user workflows:
  - User registration process
  - Login with different OAuth providers
  - Task creation and management
  - Responsive design testing
  - Cross-browser compatibility

1.4 Security Testing

- Implement security tests using safety and bandit
- Test for common vulnerabilities: (Optional)
  - SQL injection
  - XSS attacks
  - CSRF protection
  - Session hijacking
  - OAuth implementation security
- Perform dependency security scanning

2. Performance Testing

2.1 Load Testing (Optional)

- Use locust for load testing
- Test scenarios:
  - Concurrent user access
  - Multiple task creation/deletion
  - OAuth authentication under load
  - Database performance

2.2 Stress Testing

- Test application limits:
  - Maximum concurrent users
  - Database connection limits
  - Memory usage under heavy load
  - Recovery from overload

3. CI/CD Pipeline

3.1 GitHub Actions Configuration

```yaml
name: Todo App CI/CD

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

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

    steps:
    - uses: actions/checkout@v3

    - 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
        pip install pytest pytest-cov safety bandit

    - name: Run security checks
      run: |
```

```
        safety check
        bandit -r .

    - name: Run tests with coverage
      run: |
        pytest --cov=./ --cov-report=xml

    - name: Upload coverage to Codecov
      uses: codecov/codecov-action@v3

    - name: Build and push Docker image
      if: github.ref == 'refs/heads/main'
      run: |
        docker build -t todo-app .
        # Add Docker push commands here
```

3.2 Containerization

- Create Dockerfile:

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

ENV FLASK_APP=run.py
ENV FLASK_ENV=production

EXPOSE 5000

CMD ["gunicorn", "--bind", "0.0.0.0:5000", "run:app"]
```

- Create docker-compose.yml for local development:

```
version: '3.8'

services:
  web:
    build: .
```

```
    ports:
      - "5000:5000"
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/todo_db
      - SECRET_KEY=your-secret-key
    depends_on:
      - db

  db:
    image: postgres:13
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=todo_db
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

4. Documentation Requirements

4.1 Test Documentation

- Test plan and strategy
- Test cases with expected results
- Test coverage reports
- Performance test results
- Security audit reports

4.2 Setup and Deployment Documentation

- Local development setup
- Docker deployment instructions
- CI/CD pipeline configuration
- Environment variables configuration
- Database migration procedures

5. Project Deliverables

1. Complete test suite implementation
2. CI/CD pipeline configuration

3. Docker containerization setup
4. Documentation package
5. Project presentation showing:
    - Test execution and results
    - Performance metrics
    - Security analysis
    - Deployment demonstration

6. Evaluation Criteria

- Test coverage (minimum 80%)
- Successful CI/CD pipeline implementation
- Documentation quality
- Code quality and organization
- Security compliance
- Performance under load
- Successful containerization