Práctica 1

Configurar GitHub y Forkear

Git: Instalar y configurar

```
sudo apt install git
git config --global user.name "<tu nombre y apellidos entrecomillados>"
git config --global user.email <tu email>
```

Git: Generar par de claves SSH

```
ssh-keygen -t rsa -b 4096
```

Git: Visualizar clave pública y copiar

```
cd ~/.ssh
cat id_rsa.pub
```

Git: Guardar clave pública en GitHub

1. Ir a **GitHub -> Settings -> SSH and GPG keys -> New SSH key**.
2. Pegar la clave pública (`id_rsa.pub`) en el campo correspondiente.

Clonar y configurar un repositorio

1. **Clonar el repositorio original a tu máquina local:**

   ```
   git clone git@github.com:usuario-original/nombre-repositorio.git
   ```

2. **Configurar un nuevo remoto para tu fork:**

   - Crear el fork desde la interfaz de GitHub.
   - Añadir el fork como un remoto adicional:

     ```
     git remote add fork git@github.com:tu-usuario/nombre-repositorio.git
     ```

3. **Verificar los remotos configurados:**

   ```
   git remote -v
   ```

4. **Sincronizar con el repositorio original (si se actualiza):**

   ```
   git fetch origin
   git merge origin/main
   ```

5. **Subir cambios a tu fork:**

   ```
   git push fork main
   ```

Práctica 2

2.1. TODOS LOS WORKFLOWS TIENEN ESTA ESTRUCTURA MINIMA:

```yaml
 # Nombre descriptivo del workflow
name: CI/CD Workflow para pruebas automáticas

# Sección 'on': define los eventos que disparan el workflow
on:
  push:  # Se ejecuta cuando hay un push a la rama especificada
    branches:
      - main  # Solo se activa en la rama principal
  pull_request:  # También puede ejecutarse en pull requests
    branches:
      - "*"  # Aplica a todas las ramas

# Definición de los trabajos (jobs) del workflow
jobs:
  build:
    # Define el sistema operativo o entorno donde correrá el job
    runs-on: ubuntu-latest

    # Lista de pasos a realizar en el job
    steps:
      - name: Checkout del repositorio
        # Usa una acción predefinida para clonar el código
        uses: actions/checkout@v3

      - name: Configurar entorno
        # Comandos que configuran dependencias o herramientas necesarias
        run: |
          echo "Instalando dependencias..."
          sudo apt-get update && sudo apt-get install -y python3

      - name: Ejecutar pruebas
        # Ejecuta comandos para pruebas automáticas
        run: |
          echo "Ejecutando pruebas de integración..."
          python3 -m unittest discover tests

      - name: Notificar estado
        # Paso opcional para notificaciones
        run: |
          echo "Enviando resultados del build..."
```

Explicación detallada

1. **`name`**: Define el nombre del workflow que aparece en la interfaz de GitHub Actions.
   - Ejemplo: `name: CI/CD Workflow para pruebas automáticas`
2. **`on`**: Define los eventos que activan el workflow.
   - `push`: Se ejecuta cuando hay un push a las ramas especificadas.

     ```yaml
     push:
       branches:
         - main
     ```

   - `pull_request`: Actúa sobre las solicitudes de extracción en las ramas especificadas.

     ```yaml
     pull_request:
       branches:
         - "*"
     ```

3. **`jobs`**: Contiene los trabajos que se ejecutarán como parte del workflow.
   - **`build`**: Un ejemplo de trabajo que compila, prueba o valida el código.
4. **`runs-on`**: Define el sistema operativo o máquina donde se ejecutará el trabajo.
   - Ejemplo: `runs-on: ubuntu-latest`

5. **`steps`** : Lista los pasos que componen cada trabajo.
    - **`uses`** : Emplea acciones predefinidas, como `actions/checkout` para clonar el repositorio.

      ```
      uses: actions/checkout@v3
      ```

    - **`run`** : Ejecuta comandos en el entorno especificado.

      ```
      run: |
        echo "Ejecutando pruebas..."
        python3 -m unittest discover tests
      ```

2.2. Explicación Workflow: Despliegue en Pre-Producción (Webhook)

Este flujo de trabajo utiliza un webhook para desplegar automáticamente en un entorno de pre-producción después de que los workflows de "Python Lint" y "Run tests" hayan concluido con éxito. Aquí se detalla el código y su propósito:

```
name: Deploy on Webhook

on:
  workflow_run:
    workflows:
      - "Python Lint"
      - "Run tests"
    types:
      - completed

jobs:
  deploy:
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest

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

    - name: Check lint and test results
      run: |
        if [ "${{ github.event.workflow_run.conclusion }}" != "success" ]; then
          echo "Lint or Test workflow did not succeed. Exiting."
          exit 1
        fi

    - name: Trigger Deployment Webhook
      env:
        WEBHOOK_DOMAIN: ${{ secrets.WEBHOOK_DOMAIN }}
        WEBHOOK_TOKEN: ${{ secrets.WEBHOOK_TOKEN }}
      run: |
        curl -X POST https://${{ secrets.WEBHOOK_DOMAIN }}/webhook/deploy -H "Authorization: Bearer ${{ secrets.WEBHOOK_TOKEN }}"
```

Explicación detallada:

1. **Evento activador:**
    - El flujo se activa automáticamente cuando los workflows "Python Lint" y "Run tests" se completan.
    - Esto asegura que el código cumple con los estándares de calidad antes de proceder al despliegue.
2. **Condición para el despliegue:**
    - Solo se ejecuta si el branch activo es `main`:

      ```
      if: github.ref == 'refs/heads/main'
      ```

3. **Pasos del flujo:**
    - **Checkout del código:** Utiliza la acción `actions/checkout@v2` para clonar el repositorio en el runner de GitHub Actions.
    - **Verificación de resultados:** Comprueba si los workflows de linting y pruebas concluyeron con éxito:

```
if [ "${{ github.event.workflow_run.conclusion }}" != "success" ]; then
  echo "Lint or Test workflow did not succeed. Exiting."
  exit 1
fi
```

Si fallan, se detiene el flujo.

- **Despliegue mediante webhook:** Envía una solicitud POST al webhook del entorno de pre-producción:

```
curl -X POST https://${{ secrets.WEBHOOK_DOMAIN }}/webhook/deploy -H "Authorization: Bearer ${{ secrets.WEBHOOK_TOKEN }}
```

  - `WEBHOOK_DOMAIN` y `WEBHOOK_TOKEN` son secretos configurados en el repositorio para proteger el despliegue.
  - La autenticación se realiza con un token Bearer en el encabezado HTTP.

Propósito del flujo: Garantizar un despliegue seguro y automático solo si el código pasa las verificaciones de calidad.

2.3. Explicación Workflow: Despliegue en producción (Dockerhub)

Este flujo de trabajo automatiza la construcción y publicación de imágenes Docker en Docker Hub tras la creación de una nueva release. A continuación, se detalla el código y su funcionamiento:

```
name: Publish image in Docker Hub

on:
  release:
    types: [published]

jobs:
  push_to_registry:
    name: Push Docker image to Docker Hub
    runs-on: ubuntu-latest
    steps:
      - name: Check out the repo
        uses: actions/checkout@v3

      - name: Log in to Docker Hub
        uses: docker/login-action@f4ef78c080cd8ba55a85445d5b36e214a81df20a
        with:
          username: ${{ secrets.DOCKER_USER }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and push Docker image
        run: docker build --build-arg VERSION_TAG=${{ github.event.release.tag_name }} -t drorganvidez/uvlhub:${{ github.event.relea
        env:
          DOCKER_CLI_EXPERIMENTAL: enabled

      - name: Push Docker image to Docker Hub
        run: docker push drorganvidez/uvlhub:${{ github.event.release.tag_name }}

      - name: Tag and push latest
        run: |
          docker tag drorganvidez/uvlhub:${{ github.event.release.tag_name }} drorganvidez/uvlhub:latest
          docker push drorganvidez/uvlhub:latest
        env:
          DOCKER_CLI_EXPERIMENTAL: enabled
```

Explicación detallada:

1. **Evento activador:**
   - El flujo se ejecuta automáticamente cuando se publica una nueva release en GitHub:

```
on:
  release:
    types: [published]
```

2. **Pasos del flujo:**

- **Checkout del repositorio:** Usa `actions/checkout@v3` para clonar el repositorio en el runner de GitHub Actions.
- **Inicio de sesión en Docker Hub:** Utiliza `docker/login-action` para autenticar al runner con las credenciales configuradas como secretos (`DOCKER_USER` y `DOCKER_PASSWORD`).
- **Construcción y publicación de la imagen Docker:** Construye la imagen usando el archivo Dockerfile de producción y un argumento de versión basado en la etiqueta de la release:

  ```
  docker build --build-arg VERSION_TAG=${{ github.event.release.tag_name }} -t drorganvidez/uvlhub:${{ github.event.releas
  ```

  Publica la imagen etiquetada con la versión de la release en Docker Hub:

  ```
  docker push drorganvidez/uvlhub:${{ github.event.release.tag_name }}
  ```

- **Etiquetado y publicación como "latest":** Etiqueta la imagen más reciente como `latest` y la publica en Docker Hub:

  ```
  docker tag drorganvidez/uvlhub:${{ github.event.release.tag_name }} drorganvidez/uvlhub:latest
  docker push drorganvidez/uvlhub:latest
  ```

Propósito del flujo: Automatizar el proceso de creación y publicación de imágenes Docker, asegurando que cada release tenga su propia etiqueta de versión y que siempre exista una imagen etiquetada como `latest` para la versión más reciente.

2.4. Explicación Workflow: Codacy CI (sin la solución propuesta)

Este flujo de trabajo integra Codacy para realizar análisis de cobertura de código y reportar resultados al repositorio. A continuación, se detalla el código y su propósito:

```yaml
name: Codacy CI

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    services:
      mysql:
        image: mysql:5.7
        env:
          MYSQL_ROOT_PASSWORD: uvlhub_root_password
          MYSQL_DATABASE: uvlhubdb_test
          MYSQL_USER: uvlhub_user
          MYSQL_PASSWORD: uvlhub_password
        ports:
          - 3306:3306
        options: --health-cmd="mysqladmin ping" --health-interval=10s --health-timeout=5s --health-retries=3

    steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Set up Python
      uses: actions/setup-python@v5
      with:
        python-version: '3.12'

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

    - name: Upload coverage to Codacy
      run: |
        pip install codacy-coverage
        coverage run -m pytest app/modules/ --ignore-glob='*selenium*'
        coverage xml
        python-codacy-coverage -r coverage.xml
      env:
        FLASK_ENV: testing
        MARIADB_HOSTNAME: 127.0.0.1
        MARIADB_PORT: 3306
        MARIADB_TEST_DATABASE: uvlhubdb_test
        MARIADB_USER: uvlhub_user
        MARIADB_PASSWORD: uvlhub_password
        CODACY_PROJECT_TOKEN: ${{ secrets.CODACY_PROJECT_TOKEN }}
```

Explicación detallada:

1. **Evento activador:**
    - El flujo se activa en cada `push` o `pull_request` al branch `main`:

```
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
```

2. **Servicios utilizados:**
    - Se configura un servicio MySQL 5.7 para las pruebas con las credenciales definidas.
3. **Pasos del flujo:**
    - **Checkout del código:** Utiliza `actions/checkout@v4` para obtener el código del repositorio.
    - **Configuración de Python:** Configura Python 3.12 para el entorno.
    - **Instalación de dependencias:** Instala las dependencias necesarias listadas en `requirements.txt`.
    - **Cálculo y reporte de cobertura:**
        - Ejecuta pruebas con Pytest ignorando pruebas de Selenium.
        - Genera un archivo XML de cobertura.
        - Sube los resultados a Codacy utilizando el token secreto `CODACY_PROJECT_TOKEN`.

Propósito del flujo: Integrar Codacy como herramienta de análisis de cobertura y calidad de código para asegurar estándares en cada cambio al código base.

2.5. Explicación Workflow: Deploy to Render

Este flujo de trabajo realiza pruebas automatizadas y, si se completan con éxito, despliega la aplicación en Render mediante un webhook de despliegue. A continuación, se detalla el código y su funcionamiento:

```
name: Deploy to Render

on:
    push:
      tags:
      - 'v*'
    pull_request:
      branches:
        - main

jobs:

  testing:

    name: Run Tests
    runs-on: ubuntu-latest

    services:
      mysql:
        image: mysql:5.7
        env:
          MYSQL_ROOT_PASSWORD: uvlhub_root_password
          MYSQL_DATABASE: uvlhubdb_test
          MYSQL_USER: uvlhub_user
          MYSQL_PASSWORD: uvlhub_password
        ports:
          - 3306:3306
        options: --health-cmd="mysqladmin ping" --health-interval=10s --health-timeout=5s --health-retries=3

    steps:

    - uses: actions/checkout@v4

    - uses: actions/setup-python@v5
      with:
        python-version: '3.12'

    - name: Install dependencies
      run: |
```

```
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run Tests
      env:
        FLASK_ENV: testing
        MARIADB_HOSTNAME: 127.0.0.1
        MARIADB_PORT: 3306
        MARIADB_TEST_DATABASE: uvlhubdb_test
        MARIADB_USER: uvlhub_user
        MARIADB_PASSWORD: uvlhub_password
      run: |
        pytest app/modules/ --ignore-glob='*selenium*'

  deploy:
    name: Deploy to Render
    needs: testing
    runs-on: ubuntu-latest
    steps:
      - name: Check out the repo
        uses: actions/checkout@v4

      - name: Deploy to Render
        env:
          deploy_url: ${{ secrets.RENDER_DEPLOY_HOOK_URL }}
        run: |
          curl "$deploy_url"
```

Explicación detallada:

1. **Evento activador:**
   - Se activa al realizar un `push` con etiquetas que comienzan con `v` o en un `pull_request` hacia el branch `main`:

     ```
     on:
       push:
         tags:
         - 'v*'
       pull_request:
         branches:
           - main
     ```

2. **Fase de pruebas:**
   - Configura un servicio MySQL 5.7 con credenciales y opciones de salud.
   - Instala dependencias y ejecuta pruebas automatizadas con Pytest, excluyendo pruebas basadas en Selenium.
3. **Fase de despliegue:**
   - Si las pruebas son exitosas, despliega la aplicación en Render utilizando un webhook de despliegue configurado con la URL almacenada como secreto en el repositorio.
   - Realiza la solicitud HTTP al endpoint de despliegue:

     ```
     curl "$deploy_url"
     ```

Propósito del flujo: Garantizar que solo se despliegue la aplicación tras completar pruebas exitosas, manteniendo la integridad y calidad del código base.

Práctica 3

Comandos básicos

Añadir cambios al área de preparación

```
git add <archivo>
# Ejemplo:
git add practicandogit_a.txt
```

Confirmar cambios

```
git commit -m "mensaje del commit"
# Ejemplo:
git commit -m "feat: Añade archivo de prueba"
```

Enviar cambios al repositorio remoto

```
git push
# Ejemplo:
git push origin main
```

Descargar cambios sin fusionarlos

```
git fetch
# Ejemplo:
git fetch origin
```

Descargar y fusionar cambios

```
git pull
# Ejemplo:
git pull origin main
```

Gestión de ramas

Crear una nueva rama

```
git branch <nombre_rama>
# Ejemplo:
git branch feature/nueva_funcionalidad
```

Cambiar a una rama existente

```
git checkout <nombre_rama>
# Ejemplo:
git checkout feature/nueva_funcionalidad
```

Crear y cambiar a una nueva rama

```
git checkout -b <nombre_rama>
# Ejemplo:
git checkout -b feature/practicandogit_a
```

Subir una rama al repositorio remoto

```
git push -u origin <nombre_rama>
# Ejemplo:
git push -u origin feature/practicandogit_a
```

Eliminar una rama local

```
git branch -d <nombre_rama>
# Ejemplo:
git branch -d feature/obsoleta
```

Eliminar una rama remota

```
git push origin --delete <nombre_rama>
# Ejemplo:
git push origin --delete feature/obsoleta
```

Fusión de ramas

Fusión directa (merge)

```
git merge <rama_origen>
# Ejemplo:
git merge feature/practicandogit_a
```

Rebase

Reescribe el historial moviendo los commits a la punta de otra rama. `bash git rebase <rama_origen> # Ejemplo: git rebase main` ### Cherry-pick Aplicar un commit específico de una rama a otra. `bash git cherry-pick <commit_hash> # Ejemplo: git cherry-pick a1b2c3d`

Resolución de conflictos

Cuando dos ramas tienen cambios en la misma línea, Git genera conflictos.

Visualizar conflictos

```
<<<<<<< HEAD
cambio local
=======
cambio remoto
>>>>>>>
```

Selecciona el cambio deseado, guarda y añade el archivo al área de preparación: `bash git add <archivo> git commit -m "fix: resolve conflicts" # Ejemplo: git add practicandogit_a.txt git commit -m "fix: Fix conflicts"`

Herramientas avanzadas

Stashing

Guardar temporalmente los cambios no confirmados:

```
git stash
# Ejemplo:
git stash
```

Aplicar el último stash:

```
git stash apply
# Ejemplo:
git stash apply
```

Eliminar el último stash:

```
git stash drop
# Ejemplo:
git stash drop
```

Revertir cambios

Deshacer un commit:

```
git revert <commit_hash>
# Ejemplo:
git revert a1b2c3d
```

Resetear cambios

Volver a un commit anterior manteniendo los cambios en el área de trabajo:

```
git reset --soft <commit_hash>
# Ejemplo:
git reset --soft a1b2c3d
```

Eliminar cambios del área de preparación:

```
git reset --mixed <commit_hash>
# Ejemplo:
git reset --mixed a1b2c3d
```

Eliminar cambios del área de preparación y del árbol de trabajo:

```
git reset --hard <commit_hash>
# Ejemplo:
git reset --hard a1b2c3d
```

Ejercicios específicos de la práctica

Clonar un repositorio

```
git clone git@github.com:<usuario>/<repositorio>.git
# Ejemplo:
git clone git@github.com:usuario/practicas.git
```

Comprobar remoto

```
git remote -v
```

Crear archivo y rastrearlo

```
echo "tu_uvus" > practicandogit_a.txt
git add practicandogit_a.txt
```

Cambiar mensaje de commit

```
git commit --amend -m "feat: Add testing file. Closes #<ID>"
# Ejemplo:
git commit --amend -m "feat: Add testing file. Closes #1"
```

Sincronizar cambios remotos

```
git pull origin feature/practicandogit_a
```

Resolver conflictos

```
git add <archivo>
git commit -m "fix: Fix conflicts"
# Ejemplo:
git add practicandogit_a.txt
git commit -m "fix: Fix conflicts"
```

Cherry-pick commit

```
git cherry-pick <commit_hash>
# Ejemplo:
git cherry-pick a1b2c3d
```

Crear un hook

```
cd .git/hooks
touch commit-msg
# Ejemplo:
# Agrega el siguiente contenido al archivo:
#
# #!/bin/bash
# mensaje_commit=$(cat "$1")
# if ! [[ "$mensaje_commit" =~ ^(feat|fix|chore): ]]; then
#   echo "ERROR: El mensaje del commit debe comenzar con feat:, fix: o chore:"
#   exit 1
# fi
```

Práctica 4

Pruebas unitarias (solución)

Métodos principales de `mock`

- **`patch.object`** : Permite reemplazar un atributo de un objeto con un `mock` durante el contexto de una prueba. Es útil para simular el comportamiento de métodos o funciones.
- **`MagicMock`** : Una clase especial de `mock` que puede simular cualquier objeto, incluyendo propiedades y métodos. Es muy flexible para personalizar respuestas.
- **`assert_called_once_with`** : Comprueba que un método fue llamado una sola vez con argumentos específicos.

Descripción por secciones

1. **Fixture `test_client`** :

   ```
   @pytest.fixture(scope="module")
   def test_client(test_client):
       with test_client.application.app_context():
           pass
       yield test_client
   ```

   - Extiende el cliente de prueba para agregar un contexto específico.

2. **Test `get_all_by_user`** :

```
def test_get_all_by_user(notepad_service):
    with patch.object(notepad_service.repository, 'get_all_by_user') as mock_get_all:
        mock_notepads = [MagicMock(id=1), MagicMock(id=2)]
        mock_get_all.return_value = mock_notepads

        user_id = 1
        result = notepad_service.get_all_by_user(user_id)

        assert result == mock_notepads
        assert len(result) == 2
        mock_get_all.assert_called_once_with(user_id)
```

- **`patch.object`**: Se usa para reemplazar el método `get_all_by_user` con un `mock`.
- **`mock_get_all.return_value`**: Define lo que debe devolver el método simulado.
- **`assert_called_once_with(user_id)`**: Verifica que el método fue llamado con el argumento correcto.

3. **Test `create`**:

```
def test_create(notepad_service):
    with patch.object(notepad_service.repository, 'create') as mock_create:
        mock_notepad = MagicMock(id=1)
        mock_create.return_value = mock_notepad

        title = 'Test Notepad'
        body = 'Test Body'
        user_id = 1

        result = notepad_service.create(title=title, body=body, user_id=user_id)

        assert result == mock_notepad
        assert result.id == 1
        mock_create.assert_called_once_with(title=title, body=body, user_id=user_id)
```

- Simula el método `create`.
- **`mock_create.return_value`**: Devuelve un objeto `MagicMock` para imitar una nota creada.
- Comprueba que el método del repositorio recibe los argumentos correctos.

4. **Test `update`**:

```
def test_update(notepad_service):
    with patch.object(notepad_service.repository, 'update') as mock_update:
        mock_notepad = MagicMock(id=1)
        mock_update.return_value = mock_notepad

        notepad_id = 1
        title = 'Updated Notepad'
        body = 'Updated Body'

        result = notepad_service.update(notepad_id, title=title, body=body)

        assert result == mock_notepad
        mock_update.assert_called_once_with(notepad_id, title=title, body=body)
```

- Verifica que los cambios en una nota se procesan correctamente.

5. **Test `delete`**:

```
def test_delete(notepad_service):
    with patch.object(notepad_service.repository, 'delete') as mock_delete:
        mock_delete.return_value = True

        notepad_id = 1
        result = notepad_service.delete(notepad_id)

        assert result is True
        mock_delete.assert_called_once_with(notepad_id)
```

- **`mock_delete.return_value`**: Define que el método simulado devuelva `True`.
- Comprueba que el método de eliminación recibe el ID correcto.

Pruebas de integración (solución)

Fixture `test_client`

Este fixture extiende el cliente de prueba para configurar datos específicos del módulo:

```
@pytest.fixture(scope="module")
def test_client(test_client):
    with test_client.application.app_context():
        user_test = User(email='user@example.com', password='test1234')
        db.session.add(user_test)
        db.session.commit()

        profile = UserProfile(user_id=user_test.id, name="Name", surname="Surname")
        db.session.add(profile)
        db.session.commit()

    yield test_client
```

- **Crea un usuario de prueba**: Guarda un usuario en la base de datos para realizar operaciones relacionadas con notas.
- **Asocia un perfil al usuario**: Esto representa datos adicionales del usuario.
- **`yield`** : Devuelve el cliente de prueba configurado.

Prueba: `test_get_notepad`

Prueba la funcionalidad de obtener una nota específica mediante una solicitud `GET`.

```
def test_get_notepad(test_client):
    login_response = login(test_client, "user@example.com", "test1234")
    assert login_response.status_code == 200, "Login was unsuccessful."

    response = test_client.post('/notepad/create', data={
        'title': 'Notepad2',
        'body': 'This is the body of notepad2.'
    }, follow_redirects=True)
    assert response.status_code == 200

    with test_client.application.app_context():
        from app.modules.notepad.models import Notepad
        notepad = Notepad.query.filter_by(title='Notepad2', user_id=current_user.id).first()
        assert notepad is not None, "Notepad was not found in the database."

    response = test_client.get(f'/notepad/{notepad.id}')
    assert response.status_code == 200, "The notepad detail page could not be accessed."
    assert b'Notepad2' in response.data, "The notepad title is not present on the page."

    logout(test_client)
```

- **Inicio de sesión del usuario**: Comprueba que el usuario puede iniciar sesión correctamente.
- **Creación de nota**: Envía una solicitud `POST` al endpoint `/notepad/create` y verifica la respuesta.
- **Verificación en la base de datos**: Consulta la base de datos para confirmar que la nota se creó correctamente.
- **Acceso al detalle de la nota**: Envía una solicitud `GET` para verificar que la nota es accesible.
- **Cierre de sesión**: Cierra la sesión del usuario al final.

Prueba: `test_edit_notepad`

Prueba la funcionalidad de editar una nota.

```
def test_edit_notepad(test_client):
    login_response = login(test_client, "user@example.com", "test1234")
    assert login_response.status_code == 200, "Login was unsuccessful."

    response = test_client.post('/notepad/create', data={
        'title': 'Notepad3',
        'body': 'This is the body of notepad3.'
    }, follow_redirects=True)
    assert response.status_code == 200

    with test_client.application.app_context():
        from app.modules.notepad.models import Notepad
        notepad = Notepad.query.filter_by(title='Notepad3', user_id=current_user.id).first()
        assert notepad is not None, "Notepad was not found in the database."

    response = test_client.post(f'/notepad/edit/{notepad.id}', data={
        'title': 'Notepad3 Edited',
        'body': 'This is the edited body of notepad3.'
    }, follow_redirects=True)
    assert response.status_code == 200, "The notepad could not be edited."

    with test_client.application.app_context():
        notepad = Notepad.query.get(notepad.id)
        assert notepad.title == 'Notepad3 Edited', "The notepad title was not updated."
        assert notepad.body == 'This is the edited body of notepad3.', "The notepad body was not updated."

    logout(test_client)
```

- **Creación inicial**: Verifica que se puede crear una nota antes de editarla.
- **Edición de la nota**: Envía una solicitud `POST` con los nuevos datos y valida la respuesta.
- **Confirmación en la base de datos**: Comprueba que los cambios se reflejan correctamente.

Prueba: `test_delete_notepad`

Prueba la funcionalidad de eliminar una nota.

```
def test_delete_notepad(test_client):
    login_response = login(test_client, "user@example.com", "test1234")
    assert login_response.status_code == 200, "Login was unsuccessful."

    response = test_client.post('/notepad/create', data={
        'title': 'Notepad4',
        'body': 'This is the body of notepad4.'
    }, follow_redirects=True)
    assert response.status_code == 200

    with test_client.application.app_context():
        from app.modules.notepad.models import Notepad
        notepad = Notepad.query.filter_by(title='Notepad4', user_id=current_user.id).first()
        assert notepad is not None, "Notepad was not found in the database."

    response = test_client.post(f'/notepad/delete/{notepad.id}', follow_redirects=True)
    assert response.status_code == 200, "The notepad could not be deleted."

    with test_client.application.app_context():
        notepad = Notepad.query.get(notepad.id)
        assert notepad is None, "The notepad was not deleted."

    logout(test_client)
```

- **Creación inicial**: Verifica que se puede crear una nota antes de eliminarla.
- **Eliminación de la nota**: Envía una solicitud `POST` al endpoint de eliminación y valida la respuesta.
- **Confirmación en la base de datos**: Verifica que la nota ha sido eliminada exitosamente.

Pruebas de interfaz (solución)

1. Importaciones necesarias

El código importa módulos de Selenium y otras bibliotecas para manejar la automatización:

- `selenium.webdriver` : Controla el navegador.
- `By` : Localiza elementos HTML.
- `time` : Introduce pausas en la ejecución (aunque es preferible usar `WebDriverWait`).

2. Clase `TestCreatenotepad`

La clase contiene los métodos para realizar las operaciones de prueba:

a. Configuración inicial (`setup_method`)

```
def setup_method(self, method):
  self.driver = webdriver.Chrome()
  self.vars = {}
```

- **`webdriver.Chrome()`** : Inicializa el controlador para Google Chrome.
- **`self.vars`** : Un diccionario para almacenar variables necesarias durante la prueba.

b. Limpieza final (`teardown_method`)

```
def teardown_method(self, method):
  self.driver.quit()
```

- Finaliza la sesión del navegador y libera los recursos.

c. Prueba principal (`test_createnotepad`)

```
def test_createnotepad(self):
  self.driver.get("http://localhost:5000/login")
  time.sleep(2)
  self.driver.set_window_size(912, 1011)
  self.driver.find_element(By.ID, "email").send_keys("user1@example.com")
  self.driver.find_element(By.ID, "password").send_keys("1234")
  self.driver.find_element(By.ID, "submit").click()
  time.sleep(2)
  self.driver.get("http://localhost:5000/notepad/create")
  self.driver.find_element(By.ID, "title").send_keys("n1")
  self.driver.find_element(By.ID, "body").send_keys("n1")
  self.driver.find_element(By.ID, "submit").click()
```

Este método realiza los siguientes pasos:

1. **Acceso a la página de inicio de sesión**:

   ```
   self.driver.get("http://localhost:5000/login")
   ```

   - Navega a la URL especificada.
2. **Rellenar el formulario de inicio de sesión**:

   ```
   self.driver.find_element(By.ID, "email").send_keys("user1@example.com")
   self.driver.find_element(By.ID, "password").send_keys("1234")
   self.driver.find_element(By.ID, "submit").click()
   ```

   - Localiza los campos de correo electrónico y contraseña mediante `By.ID` .
   - Envía los valores necesarios para iniciar sesión.
3. **Acceso a la página de creación de notas**:

   ```
   self.driver.get("http://localhost:5000/notepad/create")
   ```

   - Navega al endpoint para crear una nueva nota.
4. **Rellenar y enviar el formulario de creación**:

```
 self.driver.find_element(By.ID, "title").send_keys("n1")
self.driver.find_element(By.ID, "body").send_keys("n1")
self.driver.find_element(By.ID, "submit").click()
```

- Introduce los datos de la nota: un título (`n1`) y un cuerpo (`n1`).
- Envía el formulario.

Pruebas de carga (solución)

Explicación detallada del código de Locust para pruebas de carga

Este código utiliza Locust, una herramienta de pruebas de carga, para simular el comportamiento de usuarios interactuando con un sistema de notas. La clase `NotepadUser` define las acciones realizadas por cada usuario simulado, incluyendo tareas como visualizar, crear y acceder a notas específicas.

Estructura del código

1. Importaciones necesarias

El código importa las bibliotecas de Locust y otras herramientas:

- `HttpUser` : Clase base para definir usuarios HTTP simulados.
- `task` : Decorador para definir las tareas que realizará cada usuario.
- `between` : Define el tiempo de espera entre tareas.
- `random` : Genera valores aleatorios para tareas como seleccionar IDs de notas.

2. Clase `NotepadUser`

Esta clase hereda de `HttpUser` y define las interacciones simuladas:

a. Tiempo de espera entre tareas

```
wait_time = between(1, 5)
```

- Cada usuario esperará entre 1 y 5 segundos antes de ejecutar la siguiente tarea.

b. Inicio de sesión (`on_start`)

```
  def on_start(self):
      self.client.post("/login", data={
          "email": "user@example.com",
          "password": "test1234"
      })
```

- Ejecutado al inicio de cada sesión de usuario simulado.
- Envía una solicitud `POST` al endpoint de inicio de sesión para autenticar al usuario.

c. Tareas definidas con `@task`

i. Ver notas existentes

```
@task(3)
def view_notepads(self):
    with self.client.get("/notepad", catch_response=True) as response:
        if response.status_code == 200:
            print("Notepad list loaded successfully.")
        else:
            response.failure(f"Got status code {response.status_code}")
```

- **Frecuencia**: Se ejecuta con una prioridad de 3 (frecuente).
- **Acción**: Envía una solicitud `GET` al endpoint `/notepad` .
- **Validación**:
  - Verifica que el código de estado HTTP es `200` .
  - Registra errores si el estado no es exitoso.

ii. Crear una nueva nota

```
@task(2)
def create_notepad(self):
    new_notepad = {
        "title": f"Notepad created by Locust {random.randint(1, 1000)}",
        "body": "This is a test notepad created during load testing."
    }
    with self.client.post("/notepad/create", data=new_notepad, catch_response=True) as response:
        if response.status_code == 200:
            print("Notepad created successfully.")
        else:
            response.failure(f"Got status code {response.status_code}")
```

- **Frecuencia**: Se ejecuta con una prioridad de 2 (moderada).
- **Acción**: Envía una solicitud `POST` al endpoint `/notepad/create` con datos generados aleatoriamente.
- **Validación**: Comprueba el estado HTTP y registra errores en caso de fallo.

iii. Ver una nota específica

```
@task(1)
def view_specific_notepad(self):
    notepad_id = random.randint(1, 1000)
    with self.client.get(f"/notepad/{notepad_id}", catch_response=True) as response:
        if response.status_code == 200:
            print(f"Notepad {notepad_id} loaded successfully.")
        elif response.status_code == 404:
            print(f"Notepad {notepad_id} not found.")
        else:
            response.failure(f"Got status code {response.status_code}")
```

- **Frecuencia**: Se ejecuta con una prioridad de 1 (menos frecuente).
- **Acción**: Envía una solicitud `GET` a un endpoint de nota específica usando un ID aleatorio.
- **Validación**:
  - Registra mensajes para los estados `200` (exitoso) y `404` (no encontrado).
  - Marca otros códigos como fallos.

d. Fin de sesión (`on_stop`)

```
  def on_stop(self):
     self.client.get("/logout")
```

- Ejecutado al final de la sesión del usuario simulado.
- Envía una solicitud `GET` al endpoint de cierre de sesión.

Práctica 5

Antes de iniciar Docker, es necesario detener el proceso de MariaDB, ya que este utiliza por defecto el puerto 3306.

Para hacerlo, puedes ejecutar los siguientes comandos:

```
sudo lsof -i :3306 # Para ver qué proceso está usando el puerto
sudo kill <PID>   # Para matar el proceso
```

Código `Dockerfile.dev` de UVLHUB:

```
### Use an official Python runtime as a parent image
FROM python:3.12-slim

# Set this environment variable to suppress the "Running as root" warning from pip
ENV PIP_ROOT_USER_ACTION=ignore

# Install necessary packages
RUN apt-get update \
    && apt-get install -y --no-install-recommends mariadb-client \
    && apt-get install -y --no-install-recommends gcc libc-dev python3-dev libffi-dev \
    && apt-get install -y --no-install-recommends curl \
    && apt-get install -y --no-install-recommends bash \
    && apt-get install -y --no-install-recommends openrc \
    && apt-get install -y --no-install-recommends build-essential

# Install Docker CLI
RUN apt-get install -y --no-install-recommends ca-certificates gnupg lsb-release \
    && curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg \
    && echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docke
    && apt-get update \
    && apt-get install -y --no-install-recommends docker-ce-cli

# Clean up
RUN apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# Set the working directory in the container to /app
WORKDIR /app

# Copy requirements.txt to the /app working directory
COPY requirements.txt .

# Removes cache at the build stage
RUN find . -type d -name "__pycache__" -exec rm -r {} + && \
    find . -type f -name "*.pyc" -exec rm -f {} +

# Copy the wait-for-db.sh script and set execution permissions
COPY --chmod=0755 scripts/wait-for-db.sh ./scripts/

# Copy the init-testing-db.sh script and set execution permissions
COPY --chmod=0755 scripts/init-testing-db.sh ./scripts/

# Copy files
COPY rosemary/ ./rosemary
COPY setup.py ./
COPY docker/ ./docker

# Update pip
RUN pip install --no-cache-dir --upgrade pip

# Install any needed packages specified in requirements.txt
RUN pip install -r requirements.txt

# Install Rosemary
RUN pip install -e ./

# Expose port 5000 for the Flask app
EXPOSE 5000
```

1. Base de la imagen

```
FROM python:3.12-slim
```

- Se utiliza una imagen ligera basada en Python 3.12 para minimizar el tamaño de la imagen y garantizar compatibilidad con el proyecto.

2. Configuración de variables de entorno

```
ENV PIP_ROOT_USER_ACTION=ignore
```

- Suprime advertencias de `pip` relacionadas con la ejecución como usuario root.

3. Instalación de paquetes necesarios

```
RUN apt-get update \
    && apt-get install -y --no-install-recommends mariadb-client \
    && apt-get install -y --no-install-recommends gcc libc-dev python3-dev libffi-dev \
    && apt-get install -y --no-install-recommends curl \
    && apt-get install -y --no-install-recommends bash \
    && apt-get install -y --no-install-recommends openrc \
    && apt-get install -y --no-install-recommends build-essential
```

- **`mariadb-client`** : Cliente de MariaDB para interactuar con bases de datos.
- **Herramientas de compilación (`gcc`, `libc-dev`, etc.)**: Necesarias para compilar extensiones de Python y bibliotecas C.
- **Otros paquetes**: `curl`, `bash` y `openrc` son herramientas auxiliares esenciales.

4. Instalación del Docker CLI

```
RUN apt-get install -y --no-install-recommends ca-certificates gnupg lsb-release \
    && curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg \
    && echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docke
    && apt-get update \
    && apt-get install -y --no-install-recommends docker-ce-cli
```

- Instala el cliente Docker para ejecutar comandos Docker desde el contenedor.
- **Pasos**:
    1. Agrega la clave GPG de Docker y el repositorio oficial.
    2. Instala el paquete `docker-ce-cli`.

5. Limpieza de caché y listas de paquetes

```
RUN apt-get clean \
    && rm -rf /var/lib/apt/lists/*
```

- Reduce el tamaño de la imagen eliminando archivos temporales creados durante la instalación de paquetes.

6. Configuración del directorio de trabajo

```
WORKDIR /app
```

- Define el directorio base para todas las operaciones dentro del contenedor.

7. Copia de dependencias del proyecto

```
COPY requirements.txt .
```

- Copia el archivo `requirements.txt` al directorio de trabajo para instalar dependencias de Python.

8. Eliminación de archivos de caché de Python

```
RUN find . -type d -name "__pycache__" -exec rm -r {} + && \
    find . -type f -name "*.pyc" -exec rm -f {} +
```

- Limpia archivos generados por Python durante el build para mantener la imagen ligera.

9. Copia de scripts y configuración de permisos

```
COPY --chmod=0755 scripts/wait-for-db.sh ./scripts/
COPY --chmod=0755 scripts/init-testing-db.sh ./scripts/
```

- Copia y otorga permisos de ejecución a los scripts de inicialización.

10. Copia de archivos adicionales

```
COPY rosemary/ ./rosemary
COPY setup.py ./
COPY docker/ ./docker
```

- Copia código fuente, configuraciones y archivos necesarios para construir e instalar el proyecto.

11. Actualización de `pip`

```
RUN pip install --no-cache-dir --upgrade pip
```

- Actualiza `pip` a la versión más reciente para evitar problemas de compatibilidad.

12. Instalación de dependencias

```
RUN pip install -r requirements.txt
```

- Instala las bibliotecas necesarias especificadas en `requirements.txt`.

13. Instalación del proyecto

```
RUN pip install -e ./
```

- Instala el proyecto en modo editable para desarrollo.

14. Exposición del puerto de la aplicación

```
EXPOSE 5000
```

- Expone el puerto `5000` para que la aplicación Flask sea accesible desde el host.

Código docker-compose.dev de UVLHUB:

```
services:
  web:
    container_name: web_app_container
    image: drorganvidez/uvlhub:dev
    env_file:
      - ../.env
    expose:
      - "5000"
    depends_on:
      - db
    build:
      context: ../
      dockerfile: docker/images/Dockerfile.dev
    volumes:
      - ../:/app
      - /var/run/docker.sock:/var/run/docker.sock
    command: [ "sh", "-c", "sh /app/docker/entrypoints/development_entrypoint.sh" ]
    networks:
      - uvlhub_network

  db:
    container_name: mariadb_container
    env_file:
      - ../.env
    build:
      context: ../
      dockerfile: docker/images/Dockerfile.mariadb
    restart: always
    ports:
      - "3306:3306"
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - uvlhub_network

  nginx:
    container_name: nginx_web_server_container
    image: nginx:latest
    volumes:
      - ./nginx/nginx.dev.conf:/etc/nginx/nginx.conf
      - ./nginx/html:/usr/share/nginx/html
    ports:
      - "80:80"
    depends_on:
      - web
    networks:
      - uvlhub_network

volumes:
  db_data:

networks:
  uvlhub_network:
```

1. Servicios definidos

a. Servicio `web`

```
web:
  container_name: web_app_container
  image: drorganvidez/uvlhub:dev
  env_file:
    - ../.env
  expose:
    - "5000"
  depends_on:
    - db
  build:
    context: ../
    dockerfile: docker/images/Dockerfile.dev
  volumes:
    - ../:/app
    - /var/run/docker.sock:/var/run/docker.sock
  command: ["sh", "-c", "sh /app/docker/entrypoints/development_entrypoint.sh"]
  networks:
    - uvlhub_network
```

- **`container_name`** : Establece el nombre del contenedor como `web_app_container` .
- **`image`** : Usa la imagen `drorganvidez/uvlhub:dev` si está disponible.
- **`env_file`** : Carga variables de entorno desde el archivo `../.env` .
- **`expose`** : Expone el puerto `5000` dentro de la red del contenedor, sin mapearlo directamente al host.
- **`depends_on`** : Garantiza que el servicio `db` se inicie antes de `web` .
- **`build`** : Permite construir una imagen personalizada utilizando:
  - **`context`** : Ubicación base para el contexto de construcción.
  - **`dockerfile`** : Ruta al Dockerfile utilizado para construir la imagen.
- **`volumes`** :
  - Monta el directorio del proyecto en `/app` dentro del contenedor.
  - Monta el socket de Docker para que el contenedor pueda interactuar con Docker.
- **`command`** : Ejecuta un script de entrada llamado `development_entrypoint.sh` .
- **`networks`** : Conecta el servicio a la red `uvlhub_network` .

b. Servicio `db`

```
db:
  container_name: mariadb_container
  env_file:
    - ../.env
  build:
    context: ../
    dockerfile: docker/images/Dockerfile.mariadb
  restart: always
  ports:
    - "3306:3306"
  volumes:
    - db_data:/var/lib/mysql
  networks:
    - uvlhub_network
```

- **`container_name`** : Asigna el nombre `mariadb_container` al contenedor.
- **`env_file`** : Carga variables de entorno desde el archivo `../.env` .
- **`build`** : Construye una imagen personalizada para MariaDB utilizando un Dockerfile dedicado.
- **`restart`** : Garantiza que el contenedor se reinicie automáticamente si falla.
- **`ports`** : Mapea el puerto `3306` del contenedor al puerto `3306` del host.
- **`volumes`** : Almacena los datos de MariaDB en el volumen `db_data` .
- **`networks`** : Conecta el servicio a la red `uvlhub_network` .

c. Servicio `nginx`

```
nginx:
  container_name: nginx_web_server_container
  image: nginx:latest
  volumes:
    - ./nginx/nginx.dev.conf:/etc/nginx/nginx.conf
    - ./nginx/html:/usr/share/nginx/html
  ports:
    - "80:80"
  depends_on:
    - web
  networks:
    - uvlhub_network
```

- **`container_name`** : Nombra el contenedor como `nginx_web_server_container`.
- **`image`** : Utiliza la imagen oficial de NGINX en su última versión.
- **`volumes`** :
  - Monta un archivo de configuración personalizado de NGINX.
  - Monta un directorio local para servir archivos estáticos.
- **`ports`** : Mapea el puerto `80` del contenedor al puerto `80` del host.
- **`depends_on`** : Garantiza que el servicio `web` se inicie antes de `nginx`.
- **`networks`** : Conecta el servicio a la red `uvlhub_network`.

---

2. Definición de volúmenes

```
volumes:
  db_data:
```

- **`db_data`** : Volumen persistente para almacenar los datos de MariaDB. Esto permite que los datos persistan incluso si el contenedor se reinicia o elimina.

3. Definición de redes

```
networks:
  uvlhub_network:
```

- **`uvlhub_network`** : Red compartida entre los servicios `web`, `db` y `nginx`, permitiendo la comunicación interna entre ellos.

Práctica 6

Explicación código Vagrantfile de UVLHUB:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

require 'pathname'
require 'fileutils'

# Method to load environment variables from an .env file and return them as a hash
def load_env(file)
  env_vars = {}
  if File.exist?(file)
    File.readlines(file).each do |line|
      key, value = line.strip.split('=', 2)
      if key && value
        ENV[key] = value
        env_vars[key] = value
      end
    end
  else
    raise "Env file not found: #{file}"
  end
  env_vars
end
```

```
# Load .env file from the top level and get the loaded variables
env_file_path = File.expand_path("../../.env", __FILE__)
puts "Loading .env file from: #{env_file_path}"
loaded_env_vars = load_env(env_file_path)

# Print only the variables that were loaded from the .env file
loaded_env_vars.each do |key, value|
  puts "#{key}: #{value}"
end


Vagrant.configure("2") do |config|

  # Choose a base box
  config.vm.box = "ubuntu/jammy64"

  # Network configuration
  config.vm.network "forwarded_port", guest: 5000, host: 5000
  config.vm.network "forwarded_port", guest: 8089, host: 8089

  # Synced folders
  config.vm.synced_folder "../", "/vagrant"

  # Use Ansible for provisioning
  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "00_main.yml"
    ansible.extra_vars = {
      flask_app_name: ENV['FLASK_APP_NAME'],
      flask_env: ENV['FLASK_ENV'],
      domain: ENV['DOMAIN'],
      mariadb_hostname: ENV['MARIADB_HOSTNAME'],
      mariadb_port: ENV['MARIADB_PORT'],
      mariadb_database: ENV['MARIADB_DATABASE'],
      mariadb_test_database: ENV['MARIADB_TEST_DATABASE'],
      mariadb_user: ENV['MARIADB_USER'],
      mariadb_password: ENV['MARIADB_PASSWORD'],
      mariadb_root_password: ENV['MARIADB_ROOT_PASSWORD'],
      working_dir: ENV['WORKING_DIR']
    }
  end

  # Load environment variables on vagrant ssh
  config.vm.provision "shell", inline: <<-SHELL
    ENV_FILE="/vagrant/.env"
    if [ -f $ENV_FILE ]; then
      export $(cat $ENV_FILE | xargs)
      while IFS= read -r line; do
        echo "export $line" >> /etc/profile.d/vagrant_env.sh
      done < $ENV_FILE
    else
      echo "Env file not found: $ENV_FILE"
      exit 1
    fi
  SHELL

  config.vm.provision "shell", run: "always", inline: <<-SHELL
    echo 'source /etc/profile.d/vagrant_env.sh' >> /home/vagrant/.bashrc
  SHELL

  # Provider configuration
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
    vb.cpus = 4
  end
```

```
end
```

1. Carga de variables de entorno

```
require 'pathname'
require 'fileutils'

def load_env(file)
  env_vars = {}
  if File.exist?(file)
    File.readlines(file).each do |line|
      key, value = line.strip.split('=', 2)
      if key && value
        ENV[key] = value
        env_vars[key] = value
      end
    end
  else
    raise "Env file not found: #{file}"
  end
  env_vars
end

env_file_path = File.expand_path("../../.env", __FILE__)
puts "Loading .env file from: #{env_file_path}"
loaded_env_vars = load_env(env_file_path)
loaded_env_vars.each do |key, value|
  puts "#{key}: #{value}"
end
```

- **Carga de variables**:
  - Este bloque lee el archivo `.env` y carga sus variables en el entorno de la máquina virtual.
  - Las variables se imprimen en consola para confirmar su carga.

---

2. Configuración general de Vagrant

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/jammy64"
```

- **`config.vm.box`** : Selecciona `ubuntu/jammy64` como la imagen base de la máquina virtual (Ubuntu 22.04).

---

3. Configuración de red

```
  config.vm.network "forwarded_port", guest: 5000, host: 5000
  config.vm.network "forwarded_port", guest: 8089, host: 8089
```

- **Puertos redirigidos**:
  - Redirige el puerto `5000` para la aplicación Flask.
  - Redirige el puerto `8089` para servicios adicionales o pruebas.

---

4. Carpetas sincronizadas

```
  config.vm.synced_folder "../", "/vagrant"
```

- Sincroniza el directorio del proyecto local con `/vagrant` en la máquina virtual.
- Permite que los cambios realizados en el host se reflejen dentro de la máquina virtual.

---

5. Configuración de aprovisionamiento con Ansible

```
  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "00_main.yml"
    ansible.extra_vars = {
      flask_app_name: ENV['FLASK_APP_NAME'],
      flask_env: ENV['FLASK_ENV'],
      domain: ENV['DOMAIN'],
      mariadb_hostname: ENV['MARIADB_HOSTNAME'],
      mariadb_port: ENV['MARIADB_PORT'],
      mariadb_database: ENV['MARIADB_DATABASE'],
      mariadb_test_database: ENV['MARIADB_TEST_DATABASE'],
      mariadb_user: ENV['MARIADB_USER'],
      mariadb_password: ENV['MARIADB_PASSWORD'],
      mariadb_root_password: ENV['MARIADB_ROOT_PASSWORD'],
      working_dir: ENV['WORKING_DIR']
    }
  end
```

- **`ansible.playbook`** : Ejecuta el archivo de playbook `00_main.yml` .
- **`ansible.extra_vars`** : Pasa las variables cargadas del archivo `.env` como variables de Ansible.

---

6. Carga de variables de entorno al iniciar sesión

```
  config.vm.provision "shell", inline: <<-SHELL
    ENV_FILE="/vagrant/.env"
    if [ -f $ENV_FILE ]; then
      export $(cat $ENV_FILE | xargs)
      while IFS= read -r line; do
        echo "export $line" >> /etc/profile.d/vagrant_env.sh
      done < $ENV_FILE
    else
      echo "Env file not found: $ENV_FILE"
      exit 1
    fi
  SHELL
```

- **Carga de entorno**: Lee el archivo `.env` y exporta sus variables.
- **Persistencia**: Añade las variables al archivo `/etc/profile.d/vagrant_env.sh` para que se carguen automáticamente en futuras sesiones.

Configuración adicional para `.bashrc`

```
  config.vm.provision "shell", run: "always", inline: <<-SHELL
    echo 'source /etc/profile.d/vagrant_env.sh' >> /home/vagrant/.bashrc
  SHELL
```

- **`.bashrc`** : Asegura que las variables de entorno se carguen cada vez que el usuario `vagrant` inicie sesión.

---

7. Configuración del proveedor VirtualBox

```
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
    vb.cpus = 4
  end
```

- **`vb.memory`** : Asigna 2048 MB de RAM a la máquina virtual.
- **`vb.cpus`** : Asigna 4 CPU para un mejor rendimiento.

Explicación código Vagrant de UVLHUB (Ansible):

02_install_mariadb.yml:

```
---
- hosts: all
  become: true

  tasks:
    - name: Install MariaDB Server
      apt:
        name:
          - mariadb-server
          - python3-pymysql
        update_cache: yes

    - name: Start and enable MariaDB service
      systemd:
        name: mariadb
        state: started
        enabled: yes

    - name: Check if MariaDB root password is already set
      shell: "mysql -u root -p'{{ mariadb_root_password }}' -e 'SELECT 1;'"
      register: mariadb_root_check
      failed_when: mariadb_root_check.rc not in [0, 1]
      changed_when: false
      ignore_errors: true

    - name: Set MariaDB root password if not set
      mysql_user:
        login_unix_socket: /run/mysqld/mysqld.sock
        login_user: 'root'
        login_password: ''
        name: 'root'
        password: '{{ mariadb_root_password }}'
        state: present
      when: mariadb_root_check.failed

    - name: Create .my.cnf for root
      copy:
        dest: /root/.my.cnf
        content: |
          [client]
          user=root
          password={{ mariadb_root_password }}
        owner: root
        mode: '0600'

    - name: Create SQL script
      copy:
        content: |
          CREATE DATABASE IF NOT EXISTS {{ mariadb_database }};
          CREATE DATABASE IF NOT EXISTS {{ mariadb_test_database }};
          CREATE USER IF NOT EXISTS '{{ mariadb_user }}'@'localhost' IDENTIFIED BY '{{ mariadb_password }}';
          GRANT ALL PRIVILEGES ON {{ mariadb_database }}.* TO '{{ mariadb_user }}'@'localhost';
          GRANT ALL PRIVILEGES ON {{ mariadb_test_database }}.* TO '{{ mariadb_user }}'@'localhost';
          FLUSH PRIVILEGES;
        dest: /tmp/setup.sql

    - name: Import SQL script
      command: bash -c "mysql < /tmp/setup.sql"

    - name: Remove temporary SQL script
      file:
        path: /tmp/setup.sql
        state: absent
```

Explicación detallada del archivo Ansible para configurar MariaDB

Este archivo Ansible se utiliza para automatizar la instalación y configuración de MariaDB en un servidor. A continuación se explica cada tarea y su función:

---

1. Encabezado del playbook

```
- hosts: all
  become: true
```

- **`hosts: all`**: Indica que las tareas se ejecutarán en todos los hosts especificados en el inventario.
- **`become: true`**: Ejecuta las tareas con privilegios elevados (similar a `sudo`).

---

2. Lista de tareas

a. Instalar MariaDB y dependencias

```
    - name: Install MariaDB Server
      apt:
        name:
          - mariadb-server
          - python3-pymysql
        update_cache: yes
```

- **`apt`**: Utiliza el gestor de paquetes `apt` para instalar MariaDB y `python3-pymysql`.
- **`update_cache: yes`**: Actualiza la caché de paquetes antes de instalar.

b. Iniciar y habilitar el servicio MariaDB

```
    - name: Start and enable MariaDB service
      systemd:
        name: mariadb
        state: started
        enabled: yes
```

- **`systemd`**: Asegura que el servicio de MariaDB esté iniciado y configurado para iniciarse automáticamente al arrancar el sistema.

c. Verificar si la contraseña root ya está configurada

```
    - name: Check if MariaDB root password is already set
      shell: "mysql -u root -p'{{ mariadb_root_password }}' -e 'SELECT 1;'"
      register: mariadb_root_check
      failed_when: mariadb_root_check.rc not in [0, 1]
      changed_when: false
      ignore_errors: true
```

- **`shell`**: Ejecuta un comando de shell para verificar si la contraseña del usuario `root` ya está configurada.
- **`register`**: Almacena el resultado en la variable `mariadb_root_check`.
- **`failed_when`**: Define las condiciones bajo las cuales la tarea se considera fallida.
- **`ignore_errors`**: Ignora errores para permitir que la configuración continúe.

d. Configurar la contraseña de root si no está configurada

```
    - name: Set MariaDB root password if not set
      mysql_user:
        login_unix_socket: /run/mysqld/mysqld.sock
        login_user: 'root'
        login_password: ''
        name: 'root'
        password: '{{ mariadb_root_password }}'
        state: present
      when: mariadb_root_check.failed
```

- **`mysql_user`**: Configura la contraseña del usuario `root` si no está definida.
- **`when`**: Solo se ejecuta si `mariadb_root_check` indica que la contraseña no está configurada.

e. Crear archivo `.my.cnf` para el usuario root

```
- name: Create .my.cnf for root
  copy:
    dest: /root/.my.cnf
    content: |
      [client]
      user=root
      password={{ mariadb_root_password }}
    owner: root
    mode: '0600'
```

- **`copy`**: Crea un archivo de configuración para que el usuario root pueda conectarse a MariaDB sin proporcionar la contraseña manualmente.
- **`mode`**: Establece permisos estrictos para proteger el archivo.

f. Crear un script SQL para configurar la base de datos

```
- name: Create SQL script
  copy:
    content: |
      CREATE DATABASE IF NOT EXISTS {{ mariadb_database }};
      CREATE DATABASE IF NOT EXISTS {{ mariadb_test_database }};
      CREATE USER IF NOT EXISTS '{{ mariadb_user }}'@'localhost' IDENTIFIED BY '{{ mariadb_password }}';
      GRANT ALL PRIVILEGES ON {{ mariadb_database }}.* TO '{{ mariadb_user }}'@'localhost';
      GRANT ALL PRIVILEGES ON {{ mariadb_test_database }}.* TO '{{ mariadb_user }}'@'localhost';
      FLUSH PRIVILEGES;
    dest: /tmp/setup.sql
```

- **`copy`**: Crea un archivo temporal con comandos SQL para:
  - Crear bases de datos si no existen.
  - Crear un usuario con privilegios.
  - Asignar privilegios a las bases de datos.

g. Importar el script SQL

```
- name: Import SQL script
  command: bash -c "mysql < /tmp/setup.sql"
```

- **`command`**: Ejecuta el script SQL utilizando el cliente de MariaDB.

h. Eliminar el archivo SQL temporal

```
- name: Remove temporary SQL script
  file:
    path: /tmp/setup.sql
    state: absent
```

- **`file`**: Borra el archivo temporal creado anteriormente.

Variables esperadas

Este playbook utiliza variables para personalizar la configuración:

- **`mariadb_root_password`**: Contraseña del usuario root de MariaDB.
- **`mariadb_database`**: Nombre de la base de datos principal.
- **`mariadb_test_database`**: Nombre de la base de datos de pruebas.
- **`mariadb_user`**: Nombre del usuario de la base de datos.
- **`mariadb_password`**: Contraseña del usuario de la base de datos.

03_mariadb_scripts.yml:

```
---
- hosts: all
  become: true

  vars:
    common_environment:
      FLASK_APP_NAME: "{{ flask_app_name }}"
      FLASK_ENV: "{{ flask_env }}"
      DOMAIN: "{{ domain }}"
      MARIADB_HOSTNAME: "{{ mariadb_hostname }}"
      MARIADB_PORT: "{{ mariadb_port }}"
      MARIADB_DATABASE: "{{ mariadb_database }}"
      MARIADB_TEST_DATABASE: "{{ mariadb_test_database }}"
      MARIADB_USER: "{{ mariadb_user }}"
      MARIADB_PASSWORD: "{{ mariadb_password }}"
      MARIADB_ROOT_PASSWORD: "{{ mariadb_root_password }}"
      WORKING_DIR: "{{ working_dir }}"

  tasks:
    - name: Set permissions for wait-for-db.sh
      file:
        path: "{{ working_dir }}scripts/wait-for-db.sh"
        mode: '0755'
        state: file
      environment: "{{ common_environment }}"

    - name: Set permissions for init-testing-db.sh
      file:
        path: "{{ working_dir }}scripts/init-testing-db.sh"
        mode: '0755'
        state: file
      environment: "{{ common_environment }}"

    - name: Wait for MariaDB to be ready
      shell: |
        {{ working_dir }}scripts/wait-for-db.sh
      args:
        executable: /bin/bash
      environment: "{{ common_environment }}"

    - name: Initialize the database
      shell: |
        {{ working_dir }}scripts/init-testing-db.sh
      args:
        executable: /bin/bash
      environment: "{{ common_environment }}"
```

Explicación detallada del archivo Ansible para configurar variables de entorno y scripts

Este archivo Ansible automatiza la configuración de un entorno Flask utilizando scripts auxiliares y variables de entorno para establecer una base de datos MariaDB y otros recursos.

---

1. Encabezado del playbook

```
- hosts: all
  become: true
```

- **`hosts: all`** : Indica que las tareas se ejecutarán en todos los hosts especificados en el inventario.
- **`become: true`** : Ejecuta las tareas con privilegios elevados (similar a `sudo`).

---

2. Definición de variables comunes

```
vars:
  common_environment:
    FLASK_APP_NAME: "{{ flask_app_name }}"
    FLASK_ENV: "{{ flask_env }}"
    DOMAIN: "{{ domain }}"
    MARIADB_HOSTNAME: "{{ mariadb_hostname }}"
    MARIADB_PORT: "{{ mariadb_port }}"
    MARIADB_DATABASE: "{{ mariadb_database }}"
    MARIADB_TEST_DATABASE: "{{ mariadb_test_database }}"
    MARIADB_USER: "{{ mariadb_user }}"
    MARIADB_PASSWORD: "{{ mariadb_password }}"
    MARIADB_ROOT_PASSWORD: "{{ mariadb_root_password }}"
    WORKING_DIR: "{{ working_dir }}"
```

- Define un conjunto de variables comunes que se utilizarán en las tareas.
- Estas variables incluyen configuraciones para Flask, MariaDB y el directorio de trabajo.

---

3. Lista de tareas

a. Configurar permisos para el script `wait-for-db.sh`

```
  - name: Set permissions for wait-for-db.sh
    file:
      path: "{{ working_dir }}scripts/wait-for-db.sh"
      mode: '0755'
      state: file
    environment: "{{ common_environment }}"
```

- **`file`**: Establece los permisos de ejecución para el script `wait-for-db.sh`.
- **`environment`**: Pasa las variables de entorno definidas en `common_environment` al contexto de esta tarea.

b. Configurar permisos para el script `init-testing-db.sh`

```
  - name: Set permissions for init-testing-db.sh
    file:
      path: "{{ working_dir }}scripts/init-testing-db.sh"
      mode: '0755'
      state: file
    environment: "{{ common_environment }}"
```

- Similar a la tarea anterior, configura permisos para el script `init-testing-db.sh`.

c. Esperar a que MariaDB esté lista

```
  - name: Wait for MariaDB to be ready
    shell: |
      {{ working_dir }}scripts/wait-for-db.sh
    args:
      executable: /bin/bash
    environment: "{{ common_environment }}"
```

- **`shell`**: Ejecuta el script `wait-for-db.sh` que verifica si MariaDB está disponible.
- **`args`**: Especifica el ejecutable (`/bin/bash`) para ejecutar el script.

d. Inicializar la base de datos

```
- name: Initialize the database
  shell: |
    {{ working_dir }}scripts/init-testing-db.sh
  args:
    executable: /bin/bash
  environment: "{{ common_environment }}"
```

- **`shell`** : Ejecuta el script `init-testing-db.sh`, que configura la base de datos de prueba y de producción.
- **`environment`** : Proporciona las variables necesarias para inicializar correctamente la base de datos.

Variables esperadas

Este playbook utiliza las siguientes variables para personalizar la configuración:

- **`flask_app_name`** : Nombre de la aplicación Flask.
- **`flask_env`** : Entorno de la aplicación (`development`, `production`, etc.).
- **`domain`** : Dominio del servidor.
- **`mariadb_hostname`** : Nombre del host de MariaDB.
- **`mariadb_port`** : Puerto en el que se ejecuta MariaDB.
- **`mariadb_database`** : Nombre de la base de datos principal.
- **`mariadb_test_database`** : Nombre de la base de datos de prueba.
- **`mariadb_user`** : Usuario de la base de datos.
- **`mariadb_password`** : Contraseña del usuario de la base de datos.
- **`mariadb_root_password`** : Contraseña del usuario root de MariaDB.
- **`working_dir`** : Directorio de trabajo principal donde se encuentran los scripts.

04_install_dependencies.yml:

```
- hosts: all
  become: true

  tasks:
    - name: Add deadsnakes PPA for installing Python 3.12
      apt_repository:
        repo: ppa:deadsnakes/ppa
        state: present
        update_cache: yes

    - name: Update the system and install Python 3.12 and dependencies
      apt:
        name:
          - python3.12
          - python3.12-venv
          - mariadb-client
        state: present

    - name: Install pip and setuptools for Python 3.12 from source
      shell: |
        wget https://bootstrap.pypa.io/get-pip.py -O /tmp/get-pip.py
        python3.12 /tmp/get-pip.py
      args:
        executable: /bin/bash

    - name: Upgrade pip and setuptools for Python 3.12
      shell: |
        python3.12 -m pip install --upgrade pip setuptools
      args:
        executable: /bin/bash

    - name: Set up the Python 3.12 virtual environment
      command: python3.12 -m venv {{ working_dir }}/vagrant_venv
      args:
        creates: "{{ working_dir }}/vagrant_venv/bin/activate"

    - name: Activate the virtual environment and install dependencies
      shell: |
        source {{ working_dir }}/vagrant_venv/bin/activate
        pip install --upgrade pip
        cd {{ working_dir }}
        pip install -r requirements.txt
        pip install -e ./
      args:
        executable: /bin/bash
```

1. Encabezado del playbook

```
- hosts: all
  become: true
```

- **`hosts: all`** : Aplica las tareas a todos los hosts especificados en el inventario.
- **`become: true`** : Ejecuta las tareas con privilegios elevados (similar a `sudo`).

---

2. Lista de tareas

a. Añadir el repositorio de Deadsnakes

```
- name: Add deadsnakes PPA for installing Python 3.12
  apt_repository:
    repo: ppa:deadsnakes/ppa
    state: present
    update_cache: yes
```

- **`apt_repository`** : Añade el repositorio Deadsnakes, que ofrece versiones actualizadas de Python.
- **`update_cache: yes`** : Actualiza la lista de paquetes disponibles tras añadir el repositorio.

b. Instalar Python 3.12 y dependencias

```
- name: Update the system and install Python 3.12 and dependencies
  apt:
    name:
      - python3.12
      - python3.12-venv
      - mariadb-client
    state: present
```

- **`apt`** : Instala Python 3.12, el módulo para entornos virtuales (`venv`) y el cliente MariaDB.
- **`state: present`** : Asegura que los paquetes estén instalados.

c. Instalar pip y setuptools desde fuente

```
- name: Install pip and setuptools for Python 3.12 from source
  shell: |
    wget https://bootstrap.pypa.io/get-pip.py -O /tmp/get-pip.py
    python3.12 /tmp/get-pip.py
  args:
    executable: /bin/bash
```

- **`shell`** : Descarga y ejecuta el script oficial para instalar pip y setuptools.
- **`wget`** : Descarga el script de instalación.
- **`python3.12`** : Ejecuta el script usando la versión instalada de Python.

d. Actualizar pip y setuptools

```
- name: Upgrade pip and setuptools for Python 3.12
  shell: |
    python3.12 -m pip install --upgrade pip setuptools
  args:
    executable: /bin/bash
```

- **`pip install --upgrade`** : Asegura que pip y setuptools estén en sus versiones más recientes.

e. Configurar el entorno virtual de Python

```
- name: Set up the Python 3.12 virtual environment
  command: python3.12 -m venv {{ working_dir }}/vagrant_venv
  args:
    creates: "{{ working_dir }}/vagrant_venv/bin/activate"
```

- **`command`** : Crea un entorno virtual en el directorio especificado por `working_dir` .
- **`args.creates`** : Evita recrear el entorno si ya existe.

f. Activar el entorno virtual e instalar dependencias

```
- name: Activate the virtual environment and install dependencies
  shell: |
    source {{ working_dir }}/vagrant_venv/bin/activate
    pip install --upgrade pip
    cd {{ working_dir }}
    pip install -r requirements.txt
    pip install -e ./
  args:
    executable: /bin/bash
```

- **`source`** : Activa el entorno virtual.
- **`pip install -r requirements.txt`** : Instala las dependencias listadas en el archivo `requirements.txt`.
- **`pip install -e ./`** : Instala la aplicación en modo editable.

Variables esperadas

Este playbook utiliza las siguientes variables:

- **`working_dir`** : Ruta donde se configurará el entorno virtual y se instalarán las dependencias.

05_run_app.yml:

```
---
- hosts: all
  become: true

  vars:
    common_environment:
      FLASK_APP_NAME: "{{ flask_app_name }}"
      FLASK_ENV: "{{ flask_env }}"
      DOMAIN: "{{ domain }}"
      MARIADB_HOSTNAME: "{{ mariadb_hostname }}"
      MARIADB_PORT: "{{ mariadb_port }}"
      MARIADB_DATABASE: "{{ mariadb_database }}"
      MARIADB_TEST_DATABASE: "{{ mariadb_test_database }}"
      MARIADB_USER: "{{ mariadb_user }}"
      MARIADB_PASSWORD: "{{ mariadb_password }}"
      MARIADB_ROOT_PASSWORD: "{{ mariadb_root_password }}"
      WORKING_DIR: "{{ working_dir }}"

  tasks:

    - name: Add webhook to .moduleignore
      shell: echo "webhook" > {{ working_dir }}.moduleignore
      args:
        executable: /bin/bash
      environment: "{{ common_environment }}"

    - name: Set database with Rosemary
      shell: |
        source {{ working_dir }}vagrant_venv/bin/activate
        cd {{ working_dir }}
        flask db upgrade
        rosemary db:seed -y --reset
      args:
        executable: /bin/bash
      environment: "{{ common_environment }}"

    - name: Run Flask application
      shell: |
        source {{ working_dir }}vagrant_venv/bin/activate
        cd {{ working_dir }}
        nohup flask run --host=0.0.0.0 --port=5000 --reload --debug > app.log 2>&1 &
      args:
        executable: /bin/bash
      environment: "{{ common_environment }}"
      async: 1
      poll: 0
```

1. Encabezado del playbook

```
- hosts: all
  become: true
```

- **`hosts: all`** : Aplica las tareas a todos los hosts especificados en el inventario.
- **`become: true`** : Ejecuta las tareas con privilegios elevados (similar a `sudo`).

---

2. Definición de variables comunes

```
vars:
  common_environment:
    FLASK_APP_NAME: "{{ flask_app_name }}"
    FLASK_ENV: "{{ flask_env }}"
    DOMAIN: "{{ domain }}"
    MARIADB_HOSTNAME: "{{ mariadb_hostname }}"
    MARIADB_PORT: "{{ mariadb_port }}"
    MARIADB_DATABASE: "{{ mariadb_database }}"
    MARIADB_TEST_DATABASE: "{{ mariadb_test_database }}"
    MARIADB_USER: "{{ mariadb_user }}"
    MARIADB_PASSWORD: "{{ mariadb_password }}"
    MARIADB_ROOT_PASSWORD: "{{ mariadb_root_password }}"
    WORKING_DIR: "{{ working_dir }}"
```

- Estas variables de entorno se pasan a las tareas para proporcionar configuraciones flexibles.
- Incluyen configuraciones de Flask, MariaDB y la ruta de trabajo.

---

3. Lista de tareas

a. Añadir una entrada al archivo `.moduleignore`

```
  - name: Add webhook to .moduleignore
    shell: echo "webhook" > {{ working_dir }}.moduleignore
    args:
      executable: /bin/bash
    environment: "{{ common_environment }}"
```

- **`shell`** : Añade la entrada `webhook` al archivo `.moduleignore` en el directorio especificado por `working_dir` .
- **`environment`** : Proporciona las variables de entorno comunes.

b. Configurar la base de datos con Rosemary

```
  - name: Set database with Rosemary
    shell: |
      source {{ working_dir }}vagrant_venv/bin/activate
      cd {{ working_dir }}
      flask db upgrade
      rosemary db:seed -y --reset
    args:
      executable: /bin/bash
    environment: "{{ common_environment }}"
```

- **`shell`** : Ejecuta los comandos para:
  - Activar el entorno virtual.
  - Migrar la base de datos utilizando `flask db upgrade` .
  - Poblar la base de datos con datos iniciales utilizando `rosemary db:seed` .
- **`environment`** : Proporciona variables necesarias para los scripts y comandos.

c. Ejecutar la aplicación Flask

```
  - name: Run Flask application
    shell: |
      source {{ working_dir }}vagrant_venv/bin/activate
      cd {{ working_dir }}
      nohup flask run --host=0.0.0.0 --port=5000 --reload --debug > app.log 2>&1 &
    args:
      executable: /bin/bash
    environment: "{{ common_environment }}"
    async: 1
    poll: 0
```

- **`shell`** : Ejecuta la aplicación Flask en modo desarrollo con:
  - **`--host=0.0.0.0`** : Permite accesos externos al servidor.

    - `--port=5000` : Escucha en el puerto 5000.
    - `--reload` : Recarga automáticamente cuando hay cambios en el código.
    - `--debug` : Activa el modo depuración.
    - Redirige la salida a `app.log` .
- `async: 1 y poll: 0` : Ejecuta la tarea en segundo plano sin esperar a que termine.

Variables esperadas

El playbook utiliza las siguientes variables:

- `flask_app_name` : Nombre de la aplicación Flask.
- `flask_env` : Entorno en el que se ejecuta Flask (`development` , `production` , etc.).
- `domain` : Dominio asociado al servidor.
- `mariadb_*` : Configuraciones relacionadas con MariaDB (host, puerto, bases de datos, usuario y contraseñas).
- `working_dir` : Ruta base donde se encuentran los scripts y la aplicación.

06_utilities.yml:

```
---
- hosts: all
  become: true

  tasks:

    - name: Add commands to .bashrc
      lineinfile:
        path: /home/vagrant/.bashrc
        line: '{{ item }}'
        create: yes
      with_items:
        - 'source {{ working_dir }}vagrant_venv/bin/activate'
        - 'cd {{ working_dir }}'
      when: ansible_user == 'vagrant'
```

1. Encabezado del playbook

```
- hosts: all
  become: true
```

- `hosts: all` : Aplica las tareas a todos los hosts especificados en el inventario.
- `become: true` : Ejecuta las tareas con privilegios elevados (similar a `sudo`).

---

2. Lista de tareas

Añadir comandos a `.bashrc`

```
    - name: Add commands to .bashrc
      lineinfile:
        path: /home/vagrant/.bashrc
        line: '{{ item }}'
        create: yes
      with_items:
        - 'source {{ working_dir }}vagrant_venv/bin/activate'
        - 'cd {{ working_dir }}'
      when: ansible_user == 'vagrant'
```

- `lineinfile` : Añade líneas al archivo especificado (`/home/vagrant/.bashrc`).
    - `path` : Ruta del archivo que se modificará.
    - `line` : Línea que se añadirá al archivo.
    - `create: yes` : Crea el archivo si no existe.
- `with_items` : Lista de comandos que se agregarán al archivo `.bashrc` :
    1. `source {{ working_dir }}vagrant_venv/bin/activate` : Activa el entorno virtual de Python automáticamente.
    2. `cd {{ working_dir }}` : Cambia al directorio de trabajo especificado.

- **`when`**: Solo ejecuta la tarea si el usuario de Ansible es `vagrant`.

Variables esperadas

Este playbook utiliza las siguientes variables:

- **`working_dir`**: Ruta base donde se encuentra el entorno virtual y los archivos del proyecto.