Empaquetado, Despliegue y Releases
© Universitas Europaea IMF

campus.euniv.eu © Universitas Europaea IMF
Juan Ulises PÉREZ VISAIRAS
campus.euniv.eu © Universitas Europaea IMF
Juan Ulises PÉREZ VISAIRAS

Indice

Empaquetado, Despliegue y Releases

1. Introducción al Empaquetado, Despliegue y Gestión de Releases

1.1. ¿Qué es el empaquetado, despliegue y gestión de versiones?

En el desarrollo de software, los procesos de **empaquetado, despliegue y gestión de versiones** son fundamentales para garantizar que una aplicación pueda ser distribuida, instalada y mantenida de manera eficiente. Implementar buenas prácticas en **empaquetado, despliegue y versionado** permite que los equipos de desarrollo trabajen de manera más eficiente y aseguren la entrega de software de alta calidad.

Definición y Objetivos

- **Empaquetado:** Consiste en la preparación del software en un formato adecuado para su distribución, incluyendo todas sus dependencias y configuraciones necesarias. En Python, se utilizan herramientas como **setuptools** para crear paquetes y **PyInstaller** para generar ejecutables.
- **Despliegue:** Se refiere al proceso de instalar y configurar una aplicación en un entorno de producción o prueba. Puede realizarse en **servidores locales, en la nube (AWS, Heroku)** o dentro de **contenedores Docker** para facilitar su portabilidad y escalabilidad.
- **Gestión de Versiones:** Implica el control de cambios en el código fuente y la asignación de números de versión mediante estrategias como **SemVer (Semantic Versioning)**. También incluye la gestión de ramas con **GitFlow** para coordinar el desarrollo y lanzamiento de nuevas versiones.

Importancia en el Ciclo de Vida del Software

Estos procesos son clave para:

- **Facilitar la distribución del software:** Permiten empaquetar aplicaciones con todas sus dependencias para evitar errores de instalación.
- **Garantizar estabilidad y escalabilidad:** Un despliegue bien gestionado reduce riesgos en producción y permite una mejor administración de recursos.
- **Optimizar el mantenimiento y actualización:** La gestión de versiones asegura un control preciso sobre cambios y correcciones de errores, facilitando futuras mejoras sin afectar la estabilidad del sistema.

1.2. Beneficios de una estrategia de despliegue y versionado eficiente

En el desarrollo de software, una **estrategia bien definida de despliegue y gestión de versiones** permite garantizar la estabilidad del sistema, minimizar errores y facilitar la escalabilidad. Implementar procesos eficientes en estas áreas optimiza el mantenimiento del software y mejora la experiencia del usuario final. Implementar una estrategia eficiente de **despliegue y versionado** mejora la estabilidad del software, optimiza la resolución de errores y permite escalar aplicaciones de forma controlada. Gracias a la **automatización, integración continua y herramientas de control de versiones**, se garantiza una entrega de software confiable y ágil.

Reducción de Errores en Producción:

Uno de los principales beneficios de una estrategia de despliegue y versionado bien planificada es la **reducción de errores en producción**. Un flujo de trabajo estructurado permite:

- **Detección temprana de errores:** Gracias a la integración de pruebas automatizadas en el pipeline de CI/CD, los errores se identifican antes de llegar al entorno de producción.
- **Implementación gradual de cambios:** Estrategias como **Canary Releases o Blue-Green Deployment** permiten probar nuevas versiones en un grupo reducido de usuarios antes de su lanzamiento global.
- **Rollback eficiente:** En caso de fallos, una gestión adecuada de versiones permite **retroceder a una versión estable rápidamente**, evitando interrupciones críticas en el servicio.
- **Historial de cambios documentado:** Un control de versiones estructurado con herramientas como **GitFlow** facilita la identificación de qué cambios provocaron fallos y su corrección eficiente.

Escalabilidad y Automatización en el Proceso de Entrega de Software

Una estrategia de despliegue eficiente permite que el software pueda **crecer y adaptarse** a nuevas necesidades sin afectar su estabilidad.

- **Automatización de despliegues:** Con herramientas como **Docker, Kubernetes o Jenkins**, es posible ejecutar despliegues sin intervención manual, reduciendo tiempos y errores humanos.
- **Balanceo de carga y escalabilidad automática:** En entornos en la nube (AWS, Azure, Google Cloud), se pueden escalar servicios automáticamente en función de la demanda, asegurando rendimiento óptimo.
- **Versionado semántico (SemVer):** Facilita la gestión de actualizaciones y dependencias, asegurando compatibilidad entre versiones y evitando cambios disruptivos en el software.

2. Empaquetado de Aplicaciones Python

2.1 Creación de Distribuciones en Python con setuptools

En el desarrollo de software, empaquetar y distribuir aplicaciones es fundamental para compartir código de manera eficiente. **setuptools** es la herramienta estándar en Python para **crear paquetes, gestionar dependencias y facilitar la distribución de software**. El uso de **setuptools** permite empaquetar, gestionar dependencias y distribuir paquetes en **PyPI**, facilitando la reutilización de código en proyectos Python.

Introducción a setuptools y su Uso en Empaquetado

setuptools es un conjunto de herramientas que permite:

- Crear paquetes **instalables** mediante pip.
- Definir dependencias y metadatos del paquete.
- Facilitar la distribución en repositorios como **PyPI (Python Package Index)**.

El empaquetado con setuptools es esencial para proyectos modulares y reutilizables. Una vez creado un paquete, otros desarrolladores pueden instalarlo con un simple pip install nombre_paquete.

Configuración de setup.py y setup.cfg

Para empaquetar un proyecto, es necesario configurar el archivo **setup.py**, que define metadatos como nombre, versión y dependencias del paquete.

Ejemplo de setup.py:

```
from setuptools import setup, find_packages

setup(
    name="mi_paquete",
    version="1.0.0",
    packages=find_packages(),
    install_requires=[
        "requests",  # Dependencias necesarias
        "numpy"
    ],
    author="Tu Nombre",
    description="Un paquete de ejemplo en Python",
)
```

Desde Python 3.8, se recomienda usar **setup.cfg** para definir configuraciones de manera declarativa:

```
[metadata]
name = mi_paquete
version = 1.0.0
author = Tu Nombre
description = Un paquete de ejemplo en Python

[options]
packages = find:
install_requires =
    requests
    numpy
```

Ambos archivos permiten que pip instale el paquete y resuelva dependencias automáticamente.

Distribución de Paquetes en PyPI (Python Package Index)

Para publicar un paquete en **PyPI**, se utilizan las siguientes herramientas:

1. **Crear el paquete:**

```
python setup.py sdist bdist_wheel
```

1. **Subir el paquete a PyPI con Twine:**

```
twine upload dist/*
```

1. **Instalar el paquete desde PyPI:**

```
pip install mi_paquete
```

Esto facilita la distribución del software para que cualquier usuario pueda instalarlo con pip.

2.2. Generación de Ejecutables con PyInstaller

Cuando se desarrolla un programa en **Python**, muchas veces se necesita distribuirlo a usuarios que no tienen instalado Python en sus sistemas. **PyInstaller** es una herramienta que permite convertir scripts de Python en **ejecutables independientes** para **Windows, Linux y macOS**, facilitando su distribución sin requerir instalación de dependencias manualmente. **PyInstaller** es una herramienta esencial para convertir scripts de Python en ejecutables independientes en **Windows, Linux y macOS**. Con sus opciones avanzadas, se pueden empaquetar dependencias y configurar la distribución para facilitar la ejecución del software sin necesidad de instalar Python.

Introducción a PyInstaller y su Propósito

PyInstaller es una herramienta de código abierto que empaqueta aplicaciones Python en archivos ejecutables (.exe, .app, .elf). Su objetivo principal es:

- Permitir que las aplicaciones escritas en Python puedan ejecutarse en diferentes sistemas sin necesidad de instalar Python.
- Incluir todas las dependencias y bibliotecas necesarias dentro del ejecutable.
- Generar versiones portables de los programas para que los usuarios las ejecuten directamente.

PyInstaller **analiza automáticamente** las dependencias del script y las empaqueta en un solo archivo o en una carpeta con los archivos requeridos.

Creación de Ejecutables en Windows, Linux y macOS

Para instalar PyInstaller, se usa el siguiente comando:

```
pip install pyinstaller
```

Para generar un ejecutable básico a partir de un script (mi_programa.py), se ejecuta:

```
pyinstaller --onefile mi_programa.py
```

Esto genera un ejecutable en la carpeta dist/, listo para ser distribuido.

Plataformas soportadas:

- **Windows:** Genera archivos .exe.
- **Linux:** Crea ejecutables ELF compatibles con distintas distribuciones.
- **macOS:** Permite crear archivos .app ejecutables.

Cada plataforma requiere ejecutar el empaquetado en su propio entorno, ya que los binarios generados no son universales.

Opciones Avanzadas: Empaquetado de Dependencias y Configuración de Spec Files

PyInstaller permite personalizar el empaquetado mediante archivos **.spec**, que incluyen configuraciones avanzadas como:

- **Añadir iconos personalizados:**
- pyinstaller --onefile --icon=icono.ico mi_programa.py
- **Incluir archivos adicionales (imágenes, bases de datos, configuraciones):**
- a = Analysis(['mi_programa.py'], datas=[('config.json', '.')])

Estos ajustes permiten optimizar y personalizar el ejecutable final para su distribución.

2.3. Pruebas y Validación de Paquetes

Una vez que un paquete ha sido empaquetado con **setuptools** o convertido en un ejecutable con **PyInstaller**, es fundamental validar su correcto funcionamiento antes de distribuirlo. Las pruebas y la validación garantizan que el software sea **estable, funcional y compatible** en distintos entornos. Realizar pruebas y validar paquetes es fundamental para garantizar **compatibilidad, estabilidad y funcionalidad** en distintas plataformas. Probar en entornos virtuales y automatizar la validación con **GitHub Actions o GitLab CI/CD** mejora la calidad del software y agiliza su distribución.

Verificación del Correcto Funcionamiento del Paquete

Después de empaquetar un programa, se deben realizar pruebas para verificar que:

- **El paquete se instala correctamente** con pip install paquete.whl o pip install . en un entorno limpio.
- **Las dependencias están correctamente configuradas** en requirements.txt o setup.cfg.
- **Las funciones y módulos del paquete operan como se espera**, ejecutando pruebas unitarias con **PyTest** o **unittest**.

Ejemplo de prueba de instalación y ejecución:

```
 pip install dist/mi_paquete.whl
python -c "import mi_paquete; mi_paquete.funcion_principal()"
```

Pruebas en Entornos Virtuales y Contenedores

Es recomendable probar el paquete en **entornos aislados** para evitar conflictos con otras bibliotecas.

- **Entornos virtuales (venv o conda):** Permiten probar el paquete en un entorno controlado antes de la distribución.
- **Contenedores con Docker:** Facilitan la validación en distintos sistemas sin necesidad de configuraciones adicionales.

Ejemplo de prueba en un entorno virtual:

```
 python -m venv test_env
source test_env/bin/activate  # En Windows: test_env\Scripts\activate
pip install mi_paquete.whl
```

Automatización del Empaquetado y Pruebas con GitHub Actions o GitLab CI/CD

Para evitar pruebas manuales, se pueden automatizar los tests y la generación de paquetes en plataformas como **GitHub Actions** o **GitLab CI/CD**.

Ejemplo de workflow en **GitHub Actions**:


```
 name: Test and Build Package
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install dependencies
        run: pip install -r requirements.txt
      - name: Run Tests
        run: pytest
      - name: Build Package
        run: python setup.py sdist bdist_wheel
```

Este proceso automatiza la validación y empaquetado del software en cada **cambio en el código**, asegurando una versión funcional antes de su lanzamiento.

3. Despliegue de Aplicaciones

3.1. Configuración de Servidores con Docker y Nginx

Para desplegar aplicaciones de manera eficiente, se utilizan tecnologías como **Docker y Nginx**, que facilitan la **gestión de servidores, escalabilidad y rendimiento** en entornos de producción. El uso de **Docker y Nginx** permite desplegar aplicaciones Python de manera eficiente, asegurando **portabilidad, escalabilidad y rendimiento** en entornos productivos.

Introducción a Docker: Contenedores vs. Máquinas Virtuales

Docker es una plataforma que permite empaquetar aplicaciones en **contenedores ligeros**, asegurando que funcionen en cualquier entorno sin depender de configuraciones específicas del sistema.

Diferencias entre contenedores y máquinas virtuales:

Característica	Contenedores (Docker)	Máquinas Virtuales (VMs)
Peso	Ligeros (comparten kernel)	Pesados (requieren SO completo)
Arranque	Rápido (segundos)	Lento (minutos)
Uso de recursos	Menos consumo de RAM y CPU	Mayor consumo de recursos
Escalabilidad	Fácil despliegue y replicación	Requiere más recursos para escalar

Docker es ideal para el despliegue de aplicaciones en la nube y microservicios, permitiendo que los entornos sean **portables, reproducibles y escalables**.

Creación de Imágenes Docker y Configuración de Dockerfile

Un **Dockerfile** es un script que define cómo se construirá una imagen Docker para contener una aplicación. Ejemplo de **Dockerfile** para una aplicación Python:

```
 # Usar una imagen base de Python
FROM python:3.9

# Establecer el directorio de trabajo dentro del contenedor
WORKDIR /app

# Copiar los archivos del proyecto al contenedor
COPY . /app

# Instalar dependencias
RUN pip install -r requirements.txt

# Exponer el puerto en el que correrá la aplicación
EXPOSE 8000

# Comando para ejecutar la aplicación
CMD ["python", "app.py"]
```

Para construir y ejecutar la imagen:

```
docker build -t mi_aplicacion .
docker run -p 8000:8000 mi_aplicacion
```

Configuración de Nginx como Servidor Proxy para Aplicaciones Python

Nginx es un servidor web que actúa como **proxy inverso**, permitiendo gestionar múltiples solicitudes y mejorar el rendimiento de aplicaciones desplegadas con Docker.

Ejemplo de configuración en nginx.conf:

```
 server {
    listen 80;
    server_name miapp.com;

    location / {
        proxy_pass http://localhost:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Con esta configuración, **Nginx redirige las solicitudes HTTP** al servicio de la aplicación en el puerto 8000, mejorando la seguridad y estabilidad del sistema.

3.2. Despliegue en la Nube: Heroku y AWS

El despliegue en la nube permite ejecutar aplicaciones sin preocuparse por la infraestructura física. **Plataformas como Heroku y AWS** ofrecen soluciones para gestionar aplicaciones de manera eficiente, escalable y con alta disponibilidad. El despliegue en la nube con **Heroku y AWS** facilita la gestión de aplicaciones con alta disponibilidad y escalabilidad. Mientras **Heroku** es ideal para proyectos pequeños y medianos, **AWS** permite mayor control y personalización en infraestructuras empresariales.

Introducción a PaaS (Platform as a Service) y su Importancia

PaaS (Platform as a Service) es un modelo de computación en la nube que proporciona una infraestructura lista para desplegar y gestionar aplicaciones sin necesidad de configurar servidores manualmente.

Ventajas de PaaS:

- Facilita el despliegue sin preocuparse por la administración de servidores.
- Escalabilidad automática según la demanda de usuarios.
- Integración con bases de datos, almacenamiento y herramientas de monitoreo.

Plataformas como **Heroku y AWS Elastic Beanstalk** permiten que los desarrolladores enfoquen sus esfuerzos en el código sin gestionar infraestructura compleja.

Despliegue en Heroku: Configuración con Procfile y PostgreSQL

Heroku es una plataforma PaaS que simplifica el despliegue de aplicaciones web.

Pasos para desplegar en Heroku:

1. **Instalar la CLI de Heroku:**

```
curl https://cli-assets.heroku.com/install.sh | sh
```

1. **Iniciar sesión y crear una aplicación en Heroku:**

```
heroku login
heroku create mi-app
```

1. **Definir un Procfile para ejecutar la aplicación:**

```
web: gunicorn app:app
```

1. **Configurar PostgreSQL en Heroku:**

```
heroku addons:create heroku-postgresql:hobby-dev
```

1. **Subir y desplegar la aplicación:**

```
git push heroku main
heroku open
```

Despliegue en AWS: EC2, Elastic Beanstalk y S3

AWS ofrece múltiples servicios para desplegar aplicaciones:

- **EC2 (Elastic Compute Cloud):** Permite lanzar servidores virtuales escalables para ejecutar aplicaciones.
- **Elastic Beanstalk:** Plataforma PaaS que gestiona automáticamente servidores, bases de datos y redes.
- **S3 (Simple Storage Service):** Almacenamiento en la nube para archivos estáticos y datos.

Para desplegar en **Elastic Beanstalk**, se usa la CLI de AWS:

```
eb init -p python-3.8 mi-app
eb create mi-app-env
```

Esto configura un entorno en AWS y despliega la aplicación sin necesidad de administrar servidores manualmente.

3.3. Estrategias de Despliegue Seguro y Automatizado

Implementar un **despliegue seguro y automatizado** es esencial para garantizar la estabilidad de una aplicación en producción. Mediante **CI/CD (Integración y Despliegue Continuo)**, monitoreo en tiempo real y estrategias de rollback, se pueden evitar interrupciones críticas y mejorar la experiencia del usuario. Aplicar estrategias de **despliegue continuo, monitoreo activo y rollback seguro** permite reducir riesgos y garantizar aplicaciones estables en producción.

Despliegue Continuo con CI/CD

El **Despliegue Continuo (CD)** permite actualizar aplicaciones automáticamente tras pasar pruebas en un flujo d e **Integración Continua (CI)**. Esto asegura que cada cambio en el código es validado antes de su implementación en producción.

Beneficios del CI/CD:

- **Automatización del proceso de entrega**, reduciendo errores manuales.
- **Implementación rápida de nuevas funcionalidades** sin afectar el servicio.
- **Facilidad para detectar y corregir errores tempranos** mediante pruebas automatizadas.

Herramientas utilizadas:

- **GitHub Actions / GitLab CI/CD**: Automatización de pruebas y despliegues.
- **Jenkins**: Integración continua en entornos empresariales.
- **Docker y Kubernetes**: Despliegue eficiente en contenedores escalables.

Monitoreo de Aplicaciones en Producción

Después del despliegue, es crucial monitorear la aplicación para detectar problemas en tiempo real.

Métricas clave en el monitoreo:

- **Uso de CPU y memoria:** Previene fallos por sobrecarga de recursos.
- **Tiempo de respuesta de la API:** Identifica problemas de rendimiento.
- **Errores en logs:** Permite detectar fallos críticos en la aplicación.

Herramientas de monitoreo:

- **Prometheus y Grafana**: Visualización en tiempo real de métricas de rendimiento.
- **New Relic y Datadog**: Supervisión avanzada de aplicaciones en la nube.

Manejo de Errores y Rollback en Caso de Fallos

Un **despliegue seguro** debe incluir estrategias para **revertir cambios rápidamente** en caso de fallos.

Técnicas de rollback:

- **Blue-Green Deployment:** Se mantiene una versión estable mientras la nueva versión se prueba en paralelo.
- **Canary Releases:** Se libera la actualización a un pequeño grupo de usuarios antes del lanzamiento total.
- **Feature Flags:** Permite activar o desactivar funcionalidades sin necesidad de hacer un nuevo despliegue.

4. Gestión de Versiones y Releases

4.1. Estrategias de Versionado con SemVer (Semantic Versioning)

El **versionado semántico (SemVer)** es un sistema estandarizado para asignar números de versión a un software de manera clara y predecible. Su objetivo es indicar cambios en la funcionalidad del software y facilitar la gestión de dependencias en proyectos. El uso de **SemVer** permite a los desarrolladores gestionar versiones de software de manera estructurada, facilitando la compatibilidad y evolución de los proyectos.

Introducción a SemVer y su Estructura (MAJOR.MINOR.PATCH)

SemVer sigue la estructura:

MAJOR.MINOR.PATCH

Donde:

- **MAJOR** (versión principal): Se incrementa cuando se realizan cambios incompatibles con versiones anteriores.
- **MINOR** (versión secundaria): Se incrementa cuando se agregan nuevas funcionalidades sin romper compatibilidad con versiones previas.
- **PATCH** (parche): Se incrementa cuando se corrigen errores sin afectar la funcionalidad existente.

Ejemplo:

- **1.0.0** → **2.0.0**: Cambios que rompen compatibilidad.
- **1.0.0** → **1.1.0**: Se agregan funciones nuevas sin afectar las existentes.
- **1.0.0** → **1.0.1**: Se corrige un error sin cambiar la funcionalidad.

Reglas para Incrementar Versiones Correctamente

Para mantener un control adecuado del software, se deben seguir estas reglas:

1. **Actualizar MAJOR** si se eliminan o modifican funcionalidades de manera incompatible.
2. **Actualizar MINOR** si se agregan nuevas características sin afectar el funcionamiento anterior.
3. **Actualizar PATCH** si solo se corrigen errores o se realizan mejoras menores.

Además, se pueden agregar etiquetas opcionales:

- **Pre-releases:** 1.2.0-beta.1 (indica una versión de prueba).
- **Build Metadata:** 1.2.0+20230901 (incluye información adicional de compilación).

4.2. Gestión de Ramas y Releases con GitFlow

GitFlow es un modelo de flujo de trabajo en **Git** que organiza el desarrollo de software mediante el uso de **ramas estructuradas**. Su objetivo es mejorar la gestión de versiones y releases, facilitando la integración de nuevas funcionalidades sin afectar la estabilidad del código en producción. GitFlow es una estrategia eficaz para **gestionar el desarrollo y las releases** de software, garantizando estabilidad y un flujo de trabajo bien organizado. La combinación de ramas, etiquetas y git merge permite una gestión estructurada del código en **proyectos colaborativos y escalables**.

Introducción a GitFlow: Flujo de Trabajo para el Control de Versiones

GitFlow proporciona una estrategia clara para manejar el desarrollo y el despliegue de aplicaciones. Su estructura se basa en el uso de diferentes ramas que separan el código en desarrollo del código estable, permitiendo:

- Facilitar la colaboración en equipos grandes.
- Mantener versiones estables mientras se añaden nuevas funcionalidades.
- Implementar correcciones sin afectar el código en desarrollo.

Para iniciar un flujo de trabajo con **GitFlow**, se usa el siguiente comando:

```
git flow init
```

Creación y Uso de Ramas en GitFlow

En GitFlow, se utilizan varias ramas con propósitos específicos:

- **main (producción):** Contiene la versión estable del software.
- **develop (desarrollo):** Rama donde se integran las nuevas funcionalidades antes de pasar a producción.
- **feature/* (nuevas funciones):** Ramas donde se desarrollan nuevas características sin afectar el código principal.
- **release/* (preparación de versiones):** Usadas para preparar una nueva versión estable antes de fusionarla en main.
- **hotfix/* (corrección de errores críticos):** Se crean para corregir problemas urgentes en producción y luego se fusionan en main y develop.

Ejemplo de creación de una nueva rama feature:

```
git flow feature start nueva-funcionalidad
```

Y para finalizar la funcionalidad:

```
git flow feature finish nueva-funcionalidad
```

Automatización del Versionado con Etiquetas (tags) y git merge

Para mantener un control de versiones eficiente, se utilizan **etiquetas (tags)** que identifican versiones específicas del software.

- **Crear una etiqueta de versión:**

```
git tag -a v1.0.0 -m "Versión estable 1.0.0"
git push origin v1.0.0
```

- **Fusionar una rama de release en main y develop:**

```
git flow release finish v1.0.0
```

Esto facilita la trazabilidad del código y permite volver a versiones anteriores si es necesario.

4.3. Publicación de Releases y Gestión de Cambios

La publicación de versiones en un proyecto de software es un proceso clave que **garantiza estabilidad, trazabilidad y transparencia** en los cambios realizados. Para ello, se utilizan herramientas como **changelogs, GitHub Releases y GitLab Releases**, junto con un conjunto de **pruebas y validaciones** antes del despliegue final. La gestión de releases y cambios es esencial en el desarrollo de software. Con el uso de **changelogs, GitHub/GitLab Releases y pruebas previas al lanzamiento**, se asegura que cada versión sea estable y documentada correctamente.

Creación de Changelogs y Documentación de Versiones

Un **changelog** es un registro detallado de los cambios introducidos en cada versión de un software. Su objetivo es informar a los usuarios y desarrolladores sobre:

- **Nuevas funcionalidades añadidas.**
- **Correcciones de errores.**
- **Mejoras en el rendimiento.**
- **Cambios que pueden afectar la compatibilidad (breaking changes).**

Ejemplo de changelog en formato estándar:

```
## [1.2.0] - 2023-09-15
### Añadido
- Nueva API para autenticación con OAuth2.
- Soporte para múltiples idiomas.

### Corregido
- Error en la carga de imágenes en dispositivos móviles.
- Problema de seguridad en la validación de formularios.

### Eliminado
- Función obsoleta `get_user_data()`, reemplazada por `fetch_user_info()`.
```

Mantener un changelog actualizado facilita la gestión del software y permite a los usuarios entender cómo evoluciona el proyecto.

Uso de GitHub Releases y GitLab Releases

Las plataformas **GitHub y GitLab** proporcionan herramientas para la gestión de releases, permitiendo:

- **Publicar versiones con etiquetas (tags).**
- **Adjuntar binarios o archivos de instalación.**

- **Incluir changelogs y documentación de la versión.**

Para crear una release en GitHub:

```
 git tag -a v1.2.0 -m "Versión 1.2.0 con mejoras de seguridad"
git push origin v1.2.0
```

Después, se puede gestionar la publicación desde la interfaz de GitHub Releases.

Pruebas y Validaciones Previas a la Publicación de una Nueva Versión

Antes de publicar una nueva versión, es fundamental ejecutar un conjunto de pruebas:

1. **Pruebas unitarias y de integración** para garantizar la estabilidad del código.
2. **Revisión manual y pruebas de usuario** para detectar errores de experiencia de usuario.
3. **Verificación de compatibilidad** con versiones anteriores para evitar fallos en migraciones.
4. **Automatización de pruebas y despliegue en CI/CD** para minimizar riesgos en producción.

Bibliografía y lecturas recomendadas:

- López, A. (2021). Despliegue de aplicaciones Python con Docker y AWS. Ediciones Anaya.
- García, J. (2020). Automatización y gestión de versiones con GitFlow y CI/CD. Alfaomega.
- Martínez, C. (2019). Creación y distribución de paquetes Python con setuptools y PyInstaller. Marcombo.
- Pérez, R. (2022). Estrategias de versionado y publicación de software en entornos empresariales. Ra-Ma.
- Documentación Oficial de Setuptools URL: https://setuptools.pypa.io/en/latest/
- Docker Docs - Guía de Implementación y Configuración URL: https://docs.docker.com/
- Guía Oficial de GitFlow y Estrategias de Versionado URL: https://nvie.com/posts/a-successful-git-branching-model/

Actividades prácticas

Ejercicio 23. Empaquetado y Despliegue de una Aplicación Web en la Nube

Una empresa de tecnología ha desarrollado una aplicación web en Python con Flask que permite a los usuarios administrar sus tareas diarias. La aplicación incluye autenticación de usuarios, almacenamiento en bases de datos y una API REST para gestionar las tareas.

El equipo de desarrollo necesita preparar la aplicación para su distribución y despliegue en la nube, asegurando que sea fácil de instalar, escalar y actualizar.

1. Empaquetar la aplicación en un paquete Python utilizando setuptools y generar un ejecutable con PyInstaller.
   Configurar Docker para contenerizar la aplicación y facilitar su despliegue en cualquier entorno.
   Desplegar la aplicación en Heroku o AWS Elastic Beanstalk para que sea accesible desde internet.
   Gestionar las versiones del software usando GitFlow y crear una release documentada con un changelog en GitHub Releases.

Procesando respuesta, no cierres el navegador, este proceso podría tardar unos segundos

Ejercicio 24. Empaquetado y Despliegue de una API REST en AWS

Un equipo de desarrollo ha creado una API REST en Flask que permite a los usuarios gestionar sus notas personales. La API incluye autenticación con JWT, almacenamiento en PostgreSQL y endpoints para crear, editar y eliminar notas.

El equipo necesita empaquetar, desplegar y gestionar versiones de la API de manera eficiente, asegurando que sea escalable y fácil de mantener.

1. Empaquetar la API como un paquete instalable en Python usando setuptools.
   Configurar un contenedor Docker para facilitar su despliegue en diferentes entornos.
   Implementar el despliegue en AWS Elastic Beanstalk con una base de datos PostgreSQL en RDS.
   Gestionar el versionado con SemVer y GitFlow, publicando releases en GitHub.

Procesando respuesta, no cierres el navegador, este proceso podría tardar unos segundos