# Documento de Arquitectura

## 1. Descripción de la Arquitectura del Sistema

### 1.1 Objetivos

Los objetivos de la arquitectura de este sistema son proporcionar una solución eficiente y moderna para las operaciones CRUD mediante una API REST en Python. Además, buscamos garantizar la escalabilidad, mantenibilidad y competitividad del sistema.

### 1.2 Componentes Principales

En cuanto a los componentes principales del sistema destacaremos:

- API REST en Python: Será la aplicación Python encargada de gestionar las operaciones CRUD.

- Base de Datos: Será la encargada de almacenar y persistir la información que hemos ido generando.

- Sistema de Pruebas: Incluye los scripts y configuraciones para ejecutar pruebas automatizadas y garantizar que el código modificado siga cumpliendo los estándares establecidos en las pruebas.

Relaciones y dependencias entre componentes:

- La API REST depende de la Base de Datos para almacenar y recuperar información.

- El Sistema de Pruebas interactúa con la API para evaluar su funcionalidad.

Nota:

- El testing de los endpoints se realiza contra una base de datos en memoria.

### 1.3 Servicios

API REST:

* Crear (POST): Se usa para añadir nuevos elementos a la base de datos

Pantalla de computadora con letras

Descripción generada automáticamente con confianza baja

* Leer (GET): Se usa para obtener información de la base de datos.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

* Actualizar (PUT): Se usa para modificar datos existentes en la base de datos.

Texto

Descripción generada automáticamente

* Eliminar (DELETE): Se usa para eliminar elementos de la base de datos.

Texto

Descripción generada automáticamente

### 1.4 Métodos

Implementación:

- La implementación sigue las mejores prácticas de desarrollo de API REST en Python, apoyándonos de un archivo test\_routes.py que nos garantice que todo está funcionando correctamente.

Ahora vamos a mostrar una serie de capturas del código a explicar su funcionamiento:

models.py

Captura de pantalla de un celular

Descripción generada automáticamente

From app import db, con este código importamos la instance de la base de datos (‘db’) desde el módulo llamado app.

Class Data(db.Model), definimos la clase llamada Data que hereda de db.Model, esto es una convención común al utilizar Flask-SQLAlchemy para la definición de modelos.

Establecemos id como Integer y clave primaria en la tabla de la base de datos garantizando como PK el id, que sea único para cada registro en la tabla.

Establecemos como String de longitud 100 el name en la columna de la tabla.

Con el return lo que queremos obtener es una cadena con los campos id y name, se ha usado la función \_\_repr\_\_ para facilitar la representación en cadena de las instancias de la clase.

manage.py

Texto

Descripción generada automáticamente

Con este código inicializamos una aplicación Flask que utiliza Flask-SQLAlchemy para la gestión de la base de datos.

Los imports de os, como ya sabemos, se importa para interactuar con el sistema operativo y poder acceder a variables de entorno, por otro lado, importamos la función create\_app y la instancia de la db para crear una instancia de la aplicación Flask.

Con la variable env\_name lo que hacemos es obtener el valor de la variable de entorno FLASK\_ENV utilizando os.getenv(), también creamos una instancia de la aplicación Flask pasando como argumento la variable del entorno que hemos obtenido anteriormente.

Con el siguiente código, with app iniciamos el contexto de aplicación utilizando el método app\_context(), asegurando que las operaciones realizadas dentro del bloque with tengan acceso al contexto de la aplicación, incluyendo la bd.

Db.create\_all(), crea todas las tablas definidas en los modelos de datos utilizando el método créate\_all(). Este paso es necesario para inicializar la base de datos con las tablas correspondientes al modelo.

Posteriormente creamos una instancia de la clase Data con el nombre SQL Test User, agregamos la instancia simple\_data a la sesión de la bd, hacemos commit para agregar el nuevo registro a la base de datos y por último hacemos un print de todo ha sido correcto.

En el punto anterior hemos mostrado los distintos endpoints, por lo que no vamos a repetirlos, pero llegados a este punto vamos a mostrar el archivo test\_routes.py para ver como hemos creado nuestros tests.

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

Cabe destacar en este punto que se modificó el documento config.py para tener en cuenta la configuración para testin, estaba así:

Texto

Descripción generada automáticamente

Y se modificó quedando así,

Captura de pantalla de un celular

Descripción generada automáticamente

Podemos observar en la configuración del test, que usamos una base de datos en memoria para dichas pruebas.

Al probar dichos tests con :

***“python -m unittest app.test\_routes”***

Texto

Descripción generada automáticamente

Pudimos ver que nos sale una advertencia de un método deprecado, así que cambiamos dicho método para evitar el mensaje de alerta, así estaba el código:Texto

Descripción generada automáticamente

Lo sustituí por este:  
Texto

Descripción generada automáticamente

Este es el resultado de los test.

Texto

Descripción generada automáticamente

### 1.5 Tecnologías Utilizadas

Las tecnologías que hemos elegido para el proyecto serán:

- Lenguaje de Programación: Python

- Base de Datos: Amazon RDS (PostrgreSQL)

- Framework: Flask

## 2. Arquitectura Cloud

### 2.1 Selección del Proveedor Cloud (AWS)

Hemos elegido AWS como proveedor de servicios en la nube debido a:

- Amplia gama de servicios que se adaptan a las necesidades del proyecto.

- Fiabilidad y escalabilidad demostradas.

- Compatibilidad con herramientas de desarrollo y despliegue.

### 2.2 Servicios AWS Utilizados

En cuanto a los servicios de AWS hemos elegido:

- EC2: Para la implementación de la API.

- RDS: Para la gestión de a base de datos PostgreSQL.

- S3: Almacenamiento de archivos estáticos si es necesario.

### 2.3 Interrelación entre Servicios

## (Adjunta un diagrama que ilustre cómo se interrelacionan los servicios AWS seleccionados, mostrando ejemplos de flujos de datos y comunicación entre servicios).

## 3. Descripción del Ciclo de Vida

### 3.1 Modelo de Desarrollo

El modelo de desarrollo adoptado es el enfoque ágil, utilizando el marco de trabajo Scrum. Los sprints se planifican con ciclos de desarrollo iterativos, lo que permite una respuesta rápida a los cambios y la entrega incremental de funcionalidades.

Usaremos git como control de versiones alojado en un repositorio en GitHub.

### 3.2 Modelo de Operaciones

En cuanto a los modelos de operaciones vamos a destacar:

- Despliegue: Utilizamos estrategias de despliegue continuo para garantizar entregas rápidas y confiables.

- Monitoreo: Implementamos un sistema de monitoreo para supervisar la salud del sistema y detectar posibles problemas.

- Escalabilidad: La infraestructura en la nube permite la escalabilidad horizontal para manejar aumentos de carga.

### 3.3 CI/CD (Integración Continua / Despliegue Continuo)

La implementación de CI/CD se realiza de la siguiente manera:

-Herramientas: Utilizaremos Jenkins para la integración y despliegue continuos de la aplicación.

- Flujo de trabajo:

1. Desarrolladores realizan commits en el repositorio.

2. Jenkins ejecuta pruebas automáticas.

3. Si las pruebas son exitosas, la aplicación se despliega automáticamente en el entorno de producción.