

1. Introducción

A lo largo de este curso hemos estudiado los principios y patrones que permiten diseñar software orientado a objetos de calidad: encapsulamiento, herencia, polimorfismo, los principios SOLID y un conjunto de patrones de diseño. Sin embargo, en la práctica profesional rara vez se construye una aplicación completamente desde cero. Lo más común es apoyarse en **frameworks**, que proveen una estructura base sobre la cual el desarrollador construye su aplicación.

Un framework es una **aplicación semi-construida** que puede ser extendida para resolver un problema específico¹. A diferencia de una biblioteca —que es un conjunto de funciones que el programador invoca cuando las necesita— un framework **define el flujo principal de la aplicación** e invoca al código del desarrollador en los puntos que corresponda. Esta diferencia, que a primera vista parece menor, tiene consecuencias profundas en la forma de diseñar y estructurar el software.

La siguiente tabla resume las diferencias más importantes entre ambos conceptos:

	Biblioteca	Framework
¿Quién controla el flujo?	El programador	El framework
¿Cómo se usa?	Se invoca cuando se necesita	Se extiende y se configura
¿Qué provee?	Funcionalidad puntual	Estructura y arquitectura
Ejemplos	<code>math</code> , <code>random</code> , <code>requests</code>	Flask, Django, FastAPI

Comprender qué es un framework y por qué funciona como funciona requiere entender cuatro principios conceptuales que analizaremos en las próximas secciones: la **inversión de control**, la **inversión e inyección de dependencias**, la **convención sobre configuración** y la **separación de responsabilidades** materializada en el patrón *Modelo-Vista-Controlador*. Recién después de eso podremos ver cómo se aplican concretamente en un framework web como Flask.

¹Johnson, R. E. y Foote, B. (1988). *Designing Reusable Classes*. Journal of Object-Oriented Programming, 1(2), 22–35. Este trabajo introdujo la noción de framework como una aplicación semi-construida que se adapta mediante extensiones.

2. Inversión de Control

El principio de **Inversión de Control** (*IoC*, por sus siglas en inglés) es el pilar sobre el que se sostienen todos los frameworks. En el código que hemos escrito hasta ahora, el programador tiene el control: es quien crea los objetos, quien llama a las funciones y quien decide el orden en que se ejecutan las cosas. En un framework, ese control se *invierte*: el framework es quien llama al código del desarrollador.

Este principio se resume habitualmente con una frase conocida como el **Principio de Hollywood**: “*No nos llames, nosotros te llamamos*”². Aplicado al desarrollo de software, significa que el desarrollador no invoca al framework; es el framework quien invoca al código del desarrollador cuando lo necesita.

2.1. Comparación con el flujo procedural

En un programa escrito de forma procedural, el flujo está completamente en manos del programador:

```
1 # El programador controla el flujo completo
2 def main():
3     nombre = input("Ingrese su nombre: ")
4     saludo = construir_saludo(nombre)
5     print(saludo)
6
7 def construir_saludo(nombre: str) -> str:
8     return f"Hola, {nombre}!"
9
10 if __name__ == "__main__":
11     main()
```

En cambio, con un framework, el desarrollador **declara** cómo debe responder la aplicación ante ciertos eventos, y el framework se encarga de detectar esos eventos y llamar al código correspondiente:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/saludo/<nombre>")
5 def saludar(nombre: str) -> str:
6     return f"Hola, {nombre}!"
```

²La expresión proviene de la industria cinematográfica de Hollywood y fue adoptada en el ámbito del software para describir el comportamiento de los frameworks. Fowler, M. (2005). *Inversion of Control*. <https://martinfowler.com/bliki/InversionOfControl.html>

```
7  
8 if __name__ == "__main__":  
9     app.run()
```

En el primer caso, el programador llama a `construir_saludo()` en el momento que él decide. En el segundo, el programador **registra** la función `saludar()` para que el framework la invoque cuando llegue una petición HTTP a la ruta `/saludo/<nombre>`. El flujo principal lo controla el framework; el desarrollador solo aporta las piezas específicas de su aplicación.

2.2. Consecuencias de la inversión

La inversión de control no es solo un detalle técnico: cambia por completo la forma en que se estructura una aplicación.

- **El framework define el esqueleto.** El desarrollador no decide cuándo arranca la aplicación, cuándo se procesan las peticiones ni en qué orden se ejecutan las cosas. Eso lo decide el framework.
- **El desarrollador aporta la lógica específica.** Su trabajo consiste en escribir funciones, clases y configuraciones que el framework invocará en los puntos de extensión que corresponda.
- **Se reduce el código repetitivo.** La infraestructura común (servidor HTTP, enrutamiento, manejo de errores, sesiones, etc.) ya está resuelta por el framework. El desarrollador se concentra en lo que es único de su aplicación.

Este principio se conecta directamente con lo que vimos en el apunte de patrones de diseño. El patrón **Template Method**³, por ejemplo, es una manifestación clásica de inversión de control: la clase base define el esqueleto de un algoritmo y delega en las subclases los pasos variables. Un framework hace lo mismo a mayor escala: define el flujo de la aplicación y delega en el desarrollador los detalles específicos.

³<https://refactoring.guru/design-patterns/template-method>

3. Inversión e Inyección de Dependencias

La inversión de control se apoya en un principio de diseño y una técnica concreta que ya conocemos por separado, pero que conviene revisar en el contexto de los frameworks. Nos referimos al **principio de inversión de dependencias** y a la **inyección de dependencias**.

3.1. Recordatorio del principio de inversión de dependencias

En el apunte de principios de diseño estudiamos el quinto principio SOLID: **Dependency Inversion Principle (DIP)**. Su enunciado establece que:

- Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.
- Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones⁴.

En la práctica, esto significa que en lugar de que una clase cree directamente sus dependencias concretas, debe recibirlas desde afuera. Como vimos en el ejemplo del apunte de SOLID, un **Gerente** no debería depender de la clase concreta **Operario**, sino de una abstracción como **ContratableParaTrabajar**.

3.2. La inyección de dependencias como técnica

La **inyección de dependencias** (DI, por sus siglas en inglés) es la técnica concreta que permite aplicar el DIP. Consiste en que un objeto **no construye** sus propias dependencias, sino que las **recibe** desde el exterior. Esta recepción puede ocurrir de varias maneras:

- Como argumentos del constructor.
- Como argumentos de un método.
- Como atributos que se asignan desde afuera.

⁴Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall. El principio aparece formulado en el capítulo 11.

Un ejemplo sin framework, escrito a mano, se vería así:

```
1 class RepositorioDeUsuarios(ABC):
2     @abstractmethod
3     def obtener_todos(self) -> list[Usuario]:
4         ...
5
6 class RepositorioMySQL(RepositorioDeUsuarios):
7     def obtener_todos(self) -> list[Usuario]:
8         # consulta a MySQL
9         ...
10
11 class ServicioDeUsuarios:
12     def __init__(self, repo: RepositorioDeUsuarios) -> None:
13         self.__repo = repo
14
15     def listar_activos(self) -> list[Usuario]:
16         return [u for u in self.__repo.obtener_todos() if u.activo]
```

La clase `ServicioDeUsuarios` no sabe si el repositorio es MySQL, un archivo o un servicio en línea. Simplemente recibe una abstracción y trabaja con ella. Quien construye el servicio es responsable de inyectar la implementación adecuada:

```
1 repo = RepositorioMySQL()
2 servicio = ServicioDeUsuarios(repo)
```

3.3. El rol del framework como contenedor

Un framework actúa típicamente como un **contenedor de inyección de dependencias**. Esto significa que el framework se encarga de:

- Crear los objetos que la aplicación necesita.
- Resolver las dependencias entre ellos.
- Entregarlos donde corresponda.

El desarrollador solo declara qué necesita; el framework se encarga del “cableado”⁵.

Esto se conecta con la inversión de control de la sección anterior: no solo el framework llama a nuestro código, sino que además le provee las dependencias que ese código necesita para funcionar.

⁵Fowler, M. (2004). *Inversion of Control Containers and the Dependency Injection Pattern*. <https://martinfowler.com/articles/injection.html>. En este artículo Fowler describe las distintas formas de inyección y el rol de los contenedores.

4. Convención sobre Configuración

El tercer pilar de los frameworks es el principio de **convención sobre configuración** (en inglés, *convention over configuration* o *coding by convention*). Fue popularizado por el framework Ruby on Rails a mediados de la década de 2000 y hoy es adoptado, en mayor o menor medida, por la mayoría de los frameworks modernos.

4.1. Qué propone el principio

La idea central es simple: en lugar de exigir al desarrollador que configure explícitamente cada aspecto de su aplicación, el framework establece **convenciones sensatas** como valores por defecto. Si el desarrollador sigue esas convenciones, no necesita escribir configuración alguna. Solo cuando quiere apartarse de la convención debe indicarlo explícitamente.

Un ejemplo clásico: si en un proyecto de Flask existe una carpeta llamada **templates/** en la raíz, el framework asume que allí estarán las plantillas HTML y las busca automáticamente. No hace falta decirle dónde están. Lo mismo ocurre con **static/** para archivos estáticos (CSS, JavaScript, imágenes).

4.2. Ejemplos de convenciones típicas

- **Estructura de carpetas:** `templates/`, `static/`, `tests/`.
- **Nombres de archivos:** `app.py` o `main.py` como punto de entrada.
- **Nombres de funciones:** en `pytest`, toda función que empiece con `test_` es considerada una prueba.
- **Rutas y controladores:** en frameworks MVC, el controlador `UsuariosController` se asocia a la ruta `/usuarios`.

4.3. Ventajas y riesgos

Ventajas:

- Reduce la cantidad de decisiones triviales que el desarrollador debe tomar.
- Disminuye el código de configuración, que suele ser extenso y propenso a errores.
- Facilita la lectura del código: cualquiera que conozca las convenciones del framework puede entender un proyecto ajeno rápidamente.

Riesgos:

- Si el desarrollador no conoce las convenciones, puede tener comportamientos “mágicos” difíciles de depurar.
- Las convenciones son decisiones de diseño del framework: si no coinciden con las del equipo, hay que configurarlas explícitamente.

Este principio se relaciona con lo que vimos en el apunte de principios de diseño: es, en el fondo, una aplicación del principio **YAGNI** (“You Aren’t Gonna Need It”). No escribas configuración para cosas que probablemente no vas a necesitar cambiar.

5. Separación de Responsabilidades y MVC

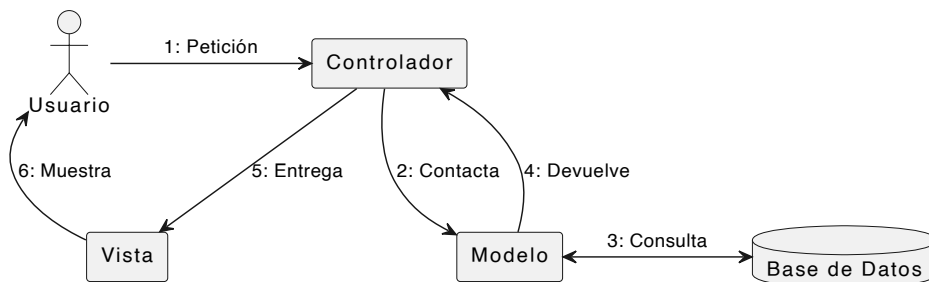
El cuarto pilar de los frameworks es la imposición de una **arquitectura** que separa responsabilidades. El principio general se conoce como **separación de responsabilidades** (SoC, por *Separation of Concerns*) y es el mismo que estudiamos cuando vimos el principio de responsabilidad única (SRP). La diferencia es que aquí se aplica a la arquitectura completa de la aplicación, no solo a una clase.

5.1. El patrón Modelo-Vista-Controlador

El patrón **Modelo-Vista-Controlador** (MVC) es la arquitectura más común en frameworks web. Su objetivo es dividir la aplicación en tres capas con responsabilidades bien definidas:

- **Modelo:** representa los datos y la lógica de negocio. Incluye las clases del dominio (por ejemplo, **Usuario**, **Producto**) y el acceso a la persistencia.

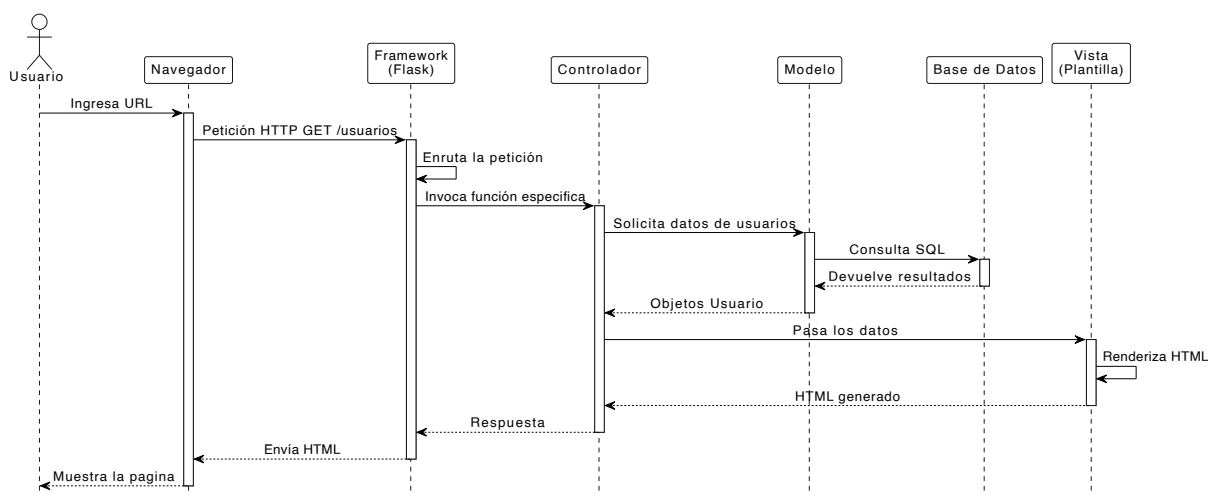
- **Vista:** se ocupa de la presentación al usuario. En el contexto web, son las plantillas HTML que se renderizan y se envían al navegador.
- **Controlador:** actúa como intermediario. Recibe las peticiones del usuario, coordina al modelo y la vista, y decide qué respuesta enviar.



Cada capa se ocupa de un aspecto distinto y se comunica con las otras a través de interfaces claras. Esto favorece la mantenibilidad, la *testabilidad*⁶ y la posibilidad de reemplazar una capa sin tocar las demás.

5.2. Flujo de una petición web

El flujo típico de una petición en un framework MVC es el siguiente:



1. El usuario realiza una petición desde el navegador (por ejemplo, GET /usuarios).
2. El framework recibe la petición y la enruta al controlador correspondiente.

⁶Anglicismo de **testability**, acuñado en el desarrollo de software, para más información: <https://www.geeksforgeeks.org/software-testing/software-testability/>

3. El controlador invoca al modelo para obtener los datos necesarios.
4. El modelo consulta la fuente de datos (base de datos, archivo, servicio externo).
5. El controlador recibe los datos y los pasa a la vista.
6. La vista genera la respuesta (HTML, JSON, etc.).
7. El framework devuelve la respuesta al navegador.

Es importante notar que el desarrollador nunca “llama” al framework: es el framework el que recibe la petición, decide a qué controlador enviarla y llama a la vista. Vemos aquí nuevamente la inversión de control en acción.

6. Un vistazo a Flask

Con los cuatro pilares ya presentados, estamos en condiciones de ver cómo se aplican en un framework web concreto. Elegimos **Flask** por su simplicidad: es un framework minimalista que provee lo esencial (enrutamiento, manejo de peticiones, plantillas) sin imponer demasiada estructura. Esto lo hace ideal para un primer contacto.

6.1. Hola mundo

El ejemplo más básico de una aplicación Flask es el siguiente:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def inicio() -> str:
6     return "¡Hola desde Flask!"
7
8 if __name__ == "__main__":
9     app.run(debug=True)
```

Analicemos qué está pasando:

- `app = Flask(__name__)` crea una instancia del framework.

- `@app.route("/")` es un **decorador** que registra la función `inicio()` para responder a las peticiones en la ruta raíz.
- El desarrollador no llama a `inicio()` directamente. Es Flask quien la invoca cuando llega una petición a `/`.
- `app.run()` arranca el servidor de desarrollo.

Aquí se ve la inversión de control: el desarrollador declara el comportamiento, el framework se encarga de ejecutarlo cuando corresponde.

6.2. Estructura típica de un proyecto

En un proyecto más grande, la convención sobre configuración se manifiesta en la organización de carpetas:

```
mi-proyecto/  
  app.py  
  templates/  
    inicio.html  
  static/  
    css/estilo.css  
    js/script.js  
  tests/  
    test_app.py
```

Flask busca automáticamente las plantillas en `templates/` y los archivos estáticos en `static/`. El desarrollador no necesita configurar nada para que esto funcione.

6.3. Los cuatro pilares en acción

A modo de síntesis, veamos cómo se manifiesta cada pilar en un proyecto Flask mínimo:

Pilar	Manifestación en Flask
Inversión de control	<code>@app.route()</code> registra funciones que Flask invoca.
Inversión de dependencias	<code>app</code> y <code>db</code> se pasan a las vistas o se importan de un módulo común.
Convención sobre configuración	<code>templates/</code> , <code>static/</code> , <code>app.py</code> .

Pilar	Manifestación en Flask
Separación de responsabilidades	Controladores como funciones de vista, modelos en módulos aparte, plantillas en <code>templates/</code> .

Este apunte es una introducción conceptual. En el siguiente estudiaremos un tipo particular de framework: el **ORM**, que se ocupa de resolver el desajuste entre el modelo de objetos y el modelo relacional de las bases de datos.

7. Referencias

Para más información puede consultar los siguientes enlaces:

- Fowler, M. (2005). *Inversion of Control*.
<https://martinfowler.com/bliki/InversionOfControl.html>
- Fowler, M. (2004). *Inversion of Control Containers and the Dependency Injection Pattern*.
<https://martinfowler.com/articles/injection.html>
- Documentación oficial de Flask:
<https://flask.palletsprojects.com/>
- Refactoring Guru:
<https://refactoring.guru/es/>