

Arquitecturas Monolíticas

Ingeniería del Software para Inteligencia Artificial

ATENCIÓN

El código de esta presentación funciona con Python 3.12
Versiones anteriores pueden requerir cambios en el código

Arquitectura de software

- Es el conjunto de decisiones estructurales sobre el diseño y organización de un sistema de software.
- Define componentes, relaciones, flujos de datos y patrones de comunicación.
- Actúa como una guía para el desarrollo de software, asegurando escalabilidad y mantenimiento.

Importancia en sistemas IA

Ejemplo

- **Mala arquitectura:** El modelo de IA está embebido en el código de la aplicación y necesita actualización manual.
- **Buena arquitectura:** El modelo es un servicio independiente en la nube, que puede reentrenarse y desplegarse sin afectar la aplicación principal.

Arquitectura monolítica

Una **arquitectura monolítica** es un modelo de desarrollo de software donde **todos los componentes de una aplicación están integrados en una sola unidad.**

La interfaz de usuario, la lógica de negocio y el acceso a datos están dentro del mismo código base y **se despliegan juntos.**

Ventajas

- **Fácil de desarrollar y probar:** No requiere integración entre múltiples servicios.
- **Menor latencia interna:** No hay comunicación externa entre servicios.
- **Menos sobrecarga técnica:** No requiere orquestación de servicios.

Desventajas

- **Difícil de escalar:** Si crece, es complicado dividirlo en partes más pequeñas.
- **Despliegues lentos:** Cualquier cambio requiere volver a desplegar toda la aplicación.
- **Menos flexible:** Si un módulo falla, puede afectar toda la aplicación.
- **Difícil de mantener** en equipos grandes, el código se vuelve complejo.

Cuándo usar

- **Para aplicaciones pequeñas o medianas** donde no se requiere escalabilidad extrema.
- **Cuando el equipo es pequeño** y no es necesario dividir responsabilidades en microservicios.
- **Para desarrollar un MVP** (Producto Mínimo Viable) rápidamente.
- **Cuando los requerimientos no cambian frecuentemente.**

Casos prácticos

Vamos a desarrollar dos casos prácticos para entender cómo implementar una arquitectura monolítica:

- **API de inferencia:** Recibe datos y devuelve predicciones de un modelo IA.
- **Backend de aplicación:** Gestiona los datos y la lógica de negocio.

API de inferencia

API de inferencia

- Vamos a implementar una aplicación que recibe un texto en español y devuelve su traducción al inglés usando un modelo de traducción preentrenado de [Hugging Face](#).
- Es un caso de uso común en sistemas de IA, donde se necesita exponer un modelo de inferencia a través de una API.
- La implementación es sencilla y no necesita estructurar demasiado el código.

API: Application Programming Interface

Comunicación con el exterior

- El software como servicio necesita comunicarse con otros servicios y aplicaciones.
- Existen distintos métodos de comunicación:
 - API REST
 - gRPC
 - WebSockets
 - Colas de mensajes (Kafka, RabbitMQ)
 - Bases de datos

Vamos a realizar la implementación de una **API REST**.

APIs REST

Recordad que una API REST debe seguir los principios RESTful:

- **Recursos:** Cada recurso tiene una URL única
- **Verbos HTTP:** GET, POST, PUT, DELETE
- **Códigos de estado:** 200, 404, 500, ...
- **Formato de datos:** JSON

Implementación de APIs

Frameworks comunes para crear APIs REST en Python:

- **Flask:** Framework ligero para crear APIs REST.
- **FastAPI:** Framework moderno y rápido para APIs REST.
- **Django REST Framework:** Extensión de Django para APIs REST.

Vamos a utilizar **FastAPI** para implementar nuestra API de inferencia.

FastAPI: Hello World

```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/")
6 def hello_world():
7     return {"message": "Hello World"}
```


FastAPI: Servicio de traducción

```
1 from fastapi import FastAPI
2 from transformers import pipeline
3
4 app = FastAPI()
5
6 @app.post("/translate")
7 def translate(prompt: str):
8     # Carga el modelo de traducción
9     pipe = pipeline("translation", model="Helsinki-NLP/opus-mt-es-en")
10    translated_text = pipe(prompt)[0]['translation_text']
11    return {"translated_text": translated_text}
```

- El valor de `prompt` se recibe en el Query String:
`http://dominio/translate?prompt=texto`
- El texto se traduce usando un modelo preentrenado.

Demo

Petición POST con Query Params

Buenas prácticas

- Carga eficiente del modelo en el arranque.

```
1 @app.post("/translate")
2 def translate(prompt: str):
3     # ✗ Carga el modelo en cada petición
4     pipe = pipeline("translation", model="Helsinki-NLP/opus-mt-es-en")
5     translated_text = pipe(prompt)[0]['translation_text']
6     return {"translated_text": translated_text}
```

Carga eficiente del modelo

```
1 from fastapi import FastAPI
2 from transformers import pipeline
3 app = FastAPI()
4
5 # ✅ Carga única del modelo al arrancar la aplicación
6 # ❌ Todavía no se gestiona la liberación de recursos
7 pipe = pipeline("translation", model="Helsinki-NLP/opus-mt-es-en")
8
9 # Añadimos este método para simplificar el endpoint
10 def translate_text(text: str) -> str:
11     if pipe is None:
12         raise RuntimeError("Translation service not initialized")
13     return pipe(text)[0]['translation_text']
14
15 @app.post("/translate")
16 def translate(prompt: str):
17     translated_text = translate_text(prompt)
18     return {"translated text": translated_text}
```

Buenas prácticas

- Carga eficiente del modelo en el arranque.
- **Gestión de recursos en memoria.**

Gestión de recursos con **lifespan**

```
1 from contextlib import asynccontextmanager
2 from fastapi import FastAPI
3 from transformers import pipeline
4
5 pipe = None
6
7 @asynccontextmanager
8 async def lifespan(app: FastAPI):
9     global pipe # Declare global to modify the outer scope
10    # Initialize the translation pipeline
11    pipe = pipeline("translation", model="Helsinki-NLP/opus-mt-es-en")
12    yield
13    # Cleanup
14    del pipe
15
16 app = FastAPI(lifespan=lifespan) # Attach lifespan handler
```

yield

- `yield` pausa la ejecución de la función. Cuando se vuelve a llamar, la función continúa desde donde se quedó.
- FastAPI llama a la función `lifespan` al arrancar la aplicación y la vuelve a llamar al cerrarla.

Inversión de control

Recuerda, muchas librerías y frameworks utilizan el patrón de **Inversión de Control** para gestionar el ciclo de vida de los recursos.

async

`async` permite que una función se ejecute de forma asíncrona, lo que es útil para tareas que pueden bloquear la ejecución principal.

Asynchronous Server Gateway Interface (ASGI)

FastAPI soporta funciones asíncronas para gestionar tareas que requieren tiempo.

[WSGI vs. ASGI](#)

Buenas prácticas

- Carga eficiente del modelo en el arranque.
- Gestión de recursos en memoria.
- **Validación de datos de entrada y salida.**

Validación de datos de entrada

- `Pydantic` permite validar los datos de forma sencilla.
- Mapea la entrada recibida como JSON en el cuerpo (*body*) de la petición HTTP.

```
1 from pydantic import BaseModel
2
3 class TranslationRequest(BaseModel):
4     text: str
5
6 @app.post("/translate")
7 async def translate(request: TranslationRequest):
8     prompt = request.text
9     translated_text = translate_text(prompt)
10    return {"translated_text": translated_text}
```

Demo

Petición POST con JSON

Validación de datos de salida

- También podemos definir la estructura de la respuesta.
- Los *type hints* documentan el código y facilitan su mantenimiento.

```
1 class TranslationResponse(BaseModel):
2     translated_text: str
3
4 @app.post("/translate", response_model=TranslationResponse)
5 async def translate(request: TranslationRequest) -> dict[str, str]:
6     prompt = request.text
7     translated_text = translate_text(prompt)
8     response = TranslationResponse(translated_text=translated_text)
9     return response
10 # return {"translation_text": translated_text} # Alternativa
```

Type hints

- Desde Python 3.9+ podemos usar `list`, `dict`, `tuple` y `set` como tipos genéricos.
- En versiones anteriores hay que usar la librería `typing`.

```
1 from typing import Dict
2
3 async def translate(request: TranslationRequest) -> Dict[str, str]:
```

Buenas prácticas

- Carga eficiente del modelo en el arranque.
- Gestión de recursos en memoria.
- Validación de datos de entrada y salida.
- **Logging para registrar eventos y errores.**

Logging

- En un servidor, es importante registrar eventos y errores para facilitar el mantenimiento y la depuración.
- Los logs se pueden **guardar en un archivo** o enviar a un servicio de monitoreo, ya que normalmente no tenemos acceso a la salida por consola.

Logging

```
1 import logging
2
3 # Create logs folder if it doesn't exist
4 os.makedirs('logs', exist_ok=True)
5
6 logging.basicConfig(level=logging.INFO, filename='../logs/translate.log', f
7 logger = logging.getLogger(__name__)
8
9 @app.post("/translate")
10 async def translate(request: TranslationRequest) -> dict[str, str]:
11     prompt = request.text
12     logger.info(f"Translating: {prompt}")
13     translated_text = translate_text(request.text)
14     response = TranslationResponse(translated_text=translated_text)
15     return response
```

Deberíamos **configurar el nivel de log** según el entorno (desarrollo: INFO/DEBUG, producción: WARNING/ERROR).

Buenas prácticas

- Carga eficiente del modelo en el arranque.
- Gestión de recursos en memoria.
- Validación de datos de entrada y salida.
- Logging para registrar eventos y errores.
- **Documentación de la API con OpenAPI.**

Documentación de los endpoints

FastAPI genera automáticamente la documentación de la API con **OpenAPI**.

Se puede completar con descripciones, ejemplos y validaciones.

- Accede a <http://localhost:8000/docs> para ver la documentación interactiva.
- Accede a <http://localhost:8000/redoc> para ver la documentación en formato ReDoc.

Esta documentación es muy útil para los desarrolladores que consumen la API.

Documentación de los endpoints

```
1  @app.post(  
2      "/translate",  
3      response_model=TranslationResponse,  
4      summary="Translate text from Spanish to English",  
5      description="""  
6      Translate text using a pre-trained model from Hugging Face:  
7      https://huggingface.co/Helsinki-NLP/opus-mt-en-es  
8      """)  
9  )  
10 async def translate(request: TranslationRequest) -> dict[str, str]:  
11     translated_text = translate_text(request.text)  
12     response = TranslationResponse(translated_text=translated_text)  
13     return response
```

Buenas prácticas

- Carga eficiente del modelo en el arranque.
- Gestión de recursos en memoria.
- Validación de datos de entrada y salida.
- Logging para registrar eventos y errores.
- Documentación de la API con OpenAPI.
- **Seguridad.**

Seguridad

- Las APIs deben protegerse contra ataques y accesos no autorizados.
 - Si son **públicos**, se deben proteger con autenticación y autorización, igual que cualquier otro servicio.
 - Si son **privados**, se pueden proteger con mediante restricciones de red o VPN (sólo accesibles para otros servicios).

Protección mediante API KEY

`db.py`: base de datos (FAKE) con usuarios y claves API.

```
1 users = [  
2     {  
3         "name": "Alice",  
4         "api_key": "5f0c7127-3be9-4488-b801-c7b6415b45e9"  
5     },  
6     {  
7         "name": "Bob",  
8         "api_key": "e54d4431-5dab-474e-b71a-0db1fcb9e659"  
9     }  
10 ]  
11  
12 # For demonstration purposes only, YOU MUST USE A DATABASE INSTEAD  
13 def get_user_from_api_key(api_key: str):  
14     for user in users:  
15         if user["api_key"] == api_key:  
16             return user  
17     return None
```

Protección mediante API KEY

`auth.py`: middleware para autenticar usuarios.

```
1 from fastapi import Security, HTTPException, status
2 from fastapi.security import APIKeyHeader
3 from db import get_user_from_api_key
4
5 api_key_header = APIKeyHeader(name="X-API-Key")
6
7 def get_user(api_key_header: str = Security(api_key_header)):
8     user = get_user_from_api_key(api_key_header)
9     if user is None:
10         raise HTTPException(
11             status_code=status.HTTP_401_UNAUTHORIZED,
12             detail="Missing or invalid API key"
13         )
14     return user
```

Protección mediante API KEY

```
1 from fastapi import Depends
2 from auth import get_user
3
4 @app.post("/secure/translate", response_model=TranslationResponse)
5 async def translate(
6     request: TranslationRequest,
7     user: dict = Depends(get_user)
8 ) -> dict[str, str]:
9
10     logger.info(f"Translation request by user {user['name']}")
11     translated_text = translate_text(request.text)
12     response = TranslationResponse(translated_text=translated_text)
13     return response
```

FastAPI permite definir **dependencias** para inyectar datos en los endpoints.

Inyección de dependencias

- Las dependencias son **funciones que se ejecutan antes de un endpoint**.
- Sirven para validar datos, autenticar usuarios, etc.
- Se pueden inyectar en los endpoints para facilitar el desarrollo y la reutilización de código.
- Pueden tener a su vez otras dependencias.

```
1 async def translate(user: dict = Depends(get_user)):  
2  
3 def get_user(api_key_header: str = Security(api_key_header)):
```

translate.py

auth.py

db.py

```
1 import os
2 import logging
3
4 from contextlib import asynccontextmanager
5 from fastapi import FastAPI, Depends
6 from pydantic import BaseModel
7
8 from transformers import pipeline
9
10 from auth import get_user
11
12
13 # Create logs folder if it doesn't exist
14 os.makedirs('logs', exist_ok=True)
15
16 logging.basicConfig(level=logging.INFO, filename='../logs/translate.log', f
17 logger = logging.getLogger(__name__)
18
```

requirements.txt

ejecución

```
1 pydantic==2.10.6
2 fastapi[standard]==0.115.10
3 torch==2.6.0
4 transformers==4.49.0
5 sentencepiece==0.2.0
6 sacremoses==0.1.1
```

async/await vs. sync (AVANZADO)

FastAPI trata internamente de forma diferente las funciones asíncronas y síncronas.

- `async def` se ejecuta en el bucle de eventos de FastAPI, usando `asyncio`.
- `def` se ejecuta en un hilo separado.

async/await vs. sync (AVANZADO)

Si llamamos a una función síncrona desde una función asíncrona, **bloqueamos el bucle de eventos**.

```
1 from fastapi import FastAPI
2 import time
3
4 app = FastAPI()
5
6 # This endpoint blocks the event loop
7 # It should be sync -> def block():
8 @app.get("/block")
9 async def block():
10     time.sleep(10)
11     return {"msg": "Sorry, I blocked the event loop"}
12
13 @app.get("/blocked")
14 async def blocked():
15     return {"msg": "I had to wait for the other endpoint to finish"}
```

async/await vs. sync (AVANZADO)

- **async/await** es más eficiente para operaciones de I/O (red, disco, etc.).
- **sync** es más eficiente para operaciones *CPU-bound* (cálculos intensivos).

async/await vs. sync (AVANZADO)

Cuando llamamos a otras funciones asíncronas se ejecutan en paralelo.

```
1 from fastapi import FastAPI
2 import time
3
4 app = FastAPI()
5
6 async def sum(a: int, b: int) -> int:
7     return a + b
8
9 @app.get("/sum")
10 async def endpoint_sum():
11     value = sum(1, 2)
12     return {"value": str(value)}
```

```
1 {"value": "<coroutine object sum at 0x791dfff090c40>"}
```

async/await vs. sync (AVANZADO)

Si necesitamos su valor de retorno, usamos `await`.

```
1 from fastapi import FastAPI
2 import time
3
4 app = FastAPI()
5
6 async def sum(a: int, b: int) -> int:
7     return a + b
8
9 @app.get("/sum")
10 async def endpoint_sum():
11     value = await sum(1, 2)
12     return {"value": str(value)}
```


Backend de aplicación

Backend de aplicación

- Vamos a implementar un backend de aplicación que gestiona los datos y la lógica de negocio.
- Es un caso de uso común en sistemas de IA, donde se necesita almacenar y procesar datos antes de enviarlos a un modelo de inferencia.
- La implementación es más compleja y necesita estructurar el código en módulos.

Una forma común de estructurar el código es siguiendo el patrón de **Arquitectura en Capas**.

Arquitectura en capas

La arquitectura en capas organiza un sistema en módulos independientes, facilitando el mantenimiento y escalabilidad.

- **Capa de Presentación (UI/API):** Interfaz para usuarios o aplicaciones.
- **Capa de Servicios:** Lógica de negocio y comunicación con el modelo de IA.
- **Capa de Datos:** Almacenamiento, gestión y preprocesamiento de datos.

Hay muchas formas de implementar una arquitectura en capas, pero esta es sencilla y efectiva.

Reglas de una arquitectura en capas

Separación de responsabilidades: Cada capa tiene una función clara y se comunican entre sí de forma jerárquica.

- **Controladores:** Reciben las peticiones del exterior.
- **Servicios:** Implementan funcionalidades.
- **Modelos:** Reflejan la estructura de las tablas.

Cada capa debe depender sólo de las capas inferiores

Organización del código

El código se organiza en distintos paquetes lógicos.

Ejemplo:

```
1  src/
2  |— main.py      # Inicia el servidor FastAPI
3  |— db/
4  |   |— db.py    # Configuración de la base de datos
5  |   |— models.py # Modelos de datos (SQLModel)
6  |— services/    # Lógica de negocio
7  |— controllers/ # Endpoints de la API
8  |— middleware/  # Autenticación, logging, etc.
9  |— util/        # Funciones auxiliares
```

Capa de datos

Capa de datos

- La **capa de datos** se encarga de gestionar el almacenamiento, acceso y preprocesamiento de los datos.
- Puede incluir bases de datos, sistemas de archivos, APIs externas, etc.
- La capa de datos ofrece una interfaz para acceder a los datos de forma sencilla y segura.

Bases de datos

- **SQL:** Bases de datos relacionales (SQLite, MySQL, PostgreSQL).
- **NoSQL:** Bases de datos no relacionales (MongoDB, Cassandra, Redis).

Bases de datos en Python

Los sistemas **ORM** (Object-Relational Mapping) facilitan el acceso a bases de datos desde Python.

Permiten **definir modelos de datos** y operar con ellos de forma sencilla, **sin necesidad de escribir SQL**.

- **SQLAlchemy**: ORM para bases de datos SQL.
- **SQLModel**: ORM para bases de datos SQL con tipado estático.
- **MongoEngine**: ORM para MongoDB.

SQLModel

- Basado en SQLAlchemy y Pydantic
- Ofrece una forma sencilla de definir modelos de datos y trabajar con bases de datos SQL.

Conexion a la base de datos

```
1 from sqlalchemy import create_engine, Session
2
3 # Cadena de conexión a la base de datos
4 DATABASE_URL = "mysql+pymysql://user:password@localhost/database"
5
6 # Crear el motor de conexión
7 engine = create_engine(DATABASE_URL, echo=True)
8
9 # Función para obtener la sesión de base de datos
10 def get_session():
11     session = Session(engine)
12     try:
13         return session # Devolvemos la sesión directamente
14     except Exception as e:
15         session.rollback() # Hacemos rollback si hay un error
16         raise e
17     finally:
18         session.close() # Nos aseguramos de cerrar la sesión cuando se sal
```

Modelos de datos

- Definen la estructura de las tablas en la base de datos.
- Son clases de Python que heredan de `SQLModel`.
- Las tablas se crean automáticamente en la base de datos.

```
1 from sqlmodel import Field, SQLModel
2
3 # Definimos un modelo de datos para la tabla 'user'
4 class User(SQLModel, table=True):
5     id: int = Field(primary_key=True)
6     name: str
7     email: str
8
9 # Se creará una tabla por cada modelo de datos
10 SQLModel.metadata.create_all(engine)
```

Modelos de datos

SQLModel se encarga de **mapear los atributos de la clase con las columnas de la tabla**, usando el tipo de datos más adecuado en cada motor de base de datos.

```
1 class User(SQLModel, table=True):  
2     id: int = Field(primary_key=True)  
3     name: str  
4     email: str
```

Columna	Tipo	Comentario
id	int(11) <i>Incremento automático</i>	
name	varchar(255)	
email	varchar(255)	

Relaciones entre modelos

- Las tablas se relacionan con **claves ajenas**.
- SQLAlchemy permite definir relaciones de forma sencilla, estableciendo una relación entre dos modelos.

```
1 class User(SQLModel, table=True):
2     id: int = Field(primary_key=True)
3     name: str = Field(index=True)
4     email: str
5     posts: list["Post"] = Relationship(back_populates="user")
6
7 class Post(SQLModel, table=True):
8     id: int = Field(primary_key=True)
9     title: str
10    content: str
11    user_id: int = Field(foreign_key="user.id")
12    user: User = Relationship(back_populates="posts")
```

Relaciones entre modelos

```
1 class User(SQLModel, table=True):
2     id: int = Field(primary_key=True)
3     name: str = Field(index=True)
4     email: str
5     posts: list["Post"] = Relationship(back_populates="user")
6
7 class Post(SQLModel, table=True):
8     id: int = Field(primary_key=True)
9     title: str
10    content: str
11    user_id: int = Field(foreign_key="user.id")
12    user: User = Relationship(back_populates="posts")
```

- `user_id`: clave ajena a la tabla `user`.
- `user`: almacenará el objeto `User` relacionado.
- `posts`: lista de objetos `Post` relacionados.

Relaciones entre modelos

Distintas formas de relacionar instancias:

```
1 with get_session() as db:
2     user = User(name="Alice", email="alice@example.com")
3     db.add(user)
4     db.commit() # Insertamos el usuario en la base de datos para obtener el
5
6     post1 = Post(user_id=user.id, title="First post", content="This is the
7     post2 = Post(user=user, title="Second post", content="This is the secon
8     post3 = Post(title="Third post", content="This is the third post")
9     post3.user = user
10    db.add_all([post1, post2, post3])
11
12    post4 = Post(title="Fourth post", content="This is the fourth post")
13    user.posts.append(post4)
14    DATABASE_URL.add(user)
15
16    db.commit()
```


Relaciones entre modelos

Las relaciones permiten obtener los objetos relacionados de forma sencilla.

```
1 from sqlmodel import select
2
3 with get_session() as db:
4     user = db.exec(select(User)).first()
5     print(user)
6     print(user.posts)
7     print(user.posts[0].title)
8
9     post = db.exec(select(Post)).first()
10    print(post)
11    print(post.user)
12    print(post.user.name)
```

Mantenimiento de la base de datos

En las primeras fases de desarrollo podemos necesitar **recrear la base de datos** para probar cambios en los modelos.

```
1 @asynccontextmanager
2 async def lifespan(app: FastAPI):
3     logging.info("Recreating database tables")
4     SQLAlchemyModel.metadata.drop_all(engine)
5     SQLAlchemyModel.metadata.create_all(engine)
6     logging.info("Seeding users")
7     seed_users()
8     logging.info("Application started")
9     yield
10    logging.info("Application shutdown")
11
12 app = FastAPI(lifespan=lifespan)
```

Mantenimiento de la base de datos

- A medida que la aplicación crece, es necesario **actualizar la estructura de la base de datos**.
- En producción no podemos borrar la base de datos, es importante **realizar migraciones** para no perder datos.
 - Cambios manuales en la estructura de la base de datos.
 - Scripts de migración para actualizar la base de datos.
 - Herramientas de migración como Alembic o Flask-Migrate.

Mantenimiento de la base de datos

- También podemos **cargar datos de prueba** en la base de datos para probar la aplicación.
- Hacemos uso de **seeders**:

```
1 @asynccontextmanager
2 async def lifespan(app: FastAPI):
3     logging.info("Recreating database tables")
4     SQLAlchemyModel.metadata.drop_all(engine)
5     SQLAlchemyModel.metadata.create_all(engine)
6     logging.info("Seeding users")
7     seed_users()
8     logging.info("Application started")
9     yield
10    logging.info("Application shutdown")
11
12 app = FastAPI(lifespan=lifespan)
```

Capa de servicios

Capa de servicios

Esta capa contiene la **lógica de negocio** y se encarga de coordinar las operaciones entre la capa de presentación y la capa de datos.

Lógica de negocio

La **lógica de negocio** es el conjunto de reglas y procesos que definen cómo funciona una aplicación.

Ejemplo: en una biblioteca, no se puede prestar un libro a un usuario en determinados casos:

- Si el libro está prestado a otro usuario.
- Si el usuario tiene el máximo de préstamos activos permitidos.
- Si el usuario tiene sanciones.

Servicios

- Los **servicios** son clases o funciones que implementan la lógica de negocio.
- Usan los modelos de datos para acceder a la base de datos.
- Ofrecen métodos para realizar operaciones de forma segura y coherente.

Servicios

- Recibimos la sesión mediante **inyección de dependencias**.
- Los métodos son **estáticos** porque no almacenan estado.

```
1 from sqlmodel import select, Session
2
3 from db import User
4
5 class UserService:
6     @staticmethod
7     def get_all(db: Session):
8         return db.exec(select(User)).all()
9
10    @staticmethod
11    def get_by_id(user_id: int, db: Session):
12        user = db.exec(select(User).where(User.id == user_id)).first()
13        if not user:
14            raise ValueError("User not found")
15        return user
```


Consultas

```
1 class UserService:
2     @staticmethod
3     def queries(db: Session):
4         # all() devuelve una lista
5         users = db.exec(select(User)).all()
6         users = db.exec(select(User).where(
7             User.email.like(f"%example.com%"))
8         ).all()
9
10        # first() devuelve el primer elemento
11        user = db.exec(select(User)).first()
12        user = db.exec(select(User).where(User.id == 1)).first()
```

Inserciones

Cualquier operación que modifique la base de datos debe confirmarse con `commit()`.

```
1 class UserService:
2     @staticmethod
3     def inserts(db: Session):
4         user = User(name="Mary", email="mary@example.com")
5         db.add(user) # Inserta un usuario
6         db.commit()  # Confirma los cambios
7
8         users = [
9             User(name="Alice", email="alice@example.com"),
10            User(name="Bob", email="bob@example.com", phone="123456789"),
11            User(name="Charlie", email="charlie@example.com", country="US")
12        ]
13        db.add_all(users) # Inserta varios usuarios
14        db.commit()       # Confirma los cambios
```

Actualizaciones y borrados

Cualquier operación que modifique la base de datos debe confirmarse con `commit()`.

```
1 class UserService:
2     @staticmethod
3     def update(db: Session):
4         user = db.exec(select(User).where(User.name == "Mary")).first()
5         user.name = "Maria" # Cambia el nombre del usuario
6         db.add(user)        # Si tiene un id, actualiza el usuario
7         db.commit()         # Confirma los cambios
8
9     @staticmethod
10    def delete(db: Session):
11        user = db.exec(select(User).where(User.name == "Maria")).first()
12        db.delete(user) # Elimina un usuario
13        db.commit()     # Confirma los cambios
```

Capa de presentación

Capa de Presentación

- Interfaz de usuario (UI) o API REST.
- Recibe peticiones y envía respuestas.
- Valida datos y autentica usuarios.
- Envía solicitudes a la capa de servicios.

Uso de métodos HTTP

Método HTTP	Uso
GET	Obtener datos (GET /users/{id})
POST	Crear un nuevo recurso (POST /users/)
PUT	Actualizar un recurso (PUT /users/{id})
DELETE	Eliminar un recurso (DELETE /users/{id})

Respuestas correctas

Código	Significado	Uso en API REST
200 OK	Éxito	Respuesta con éxito a una solicitud GET o PUT.
201 Created	Recurso creado	Se usa en respuestas de POST cuando se crea un nuevo recurso.
204 No Content	Sin contenido	La solicitud terminó con éxito pero no hay datos que devolver (ej. DELETE).

Códigos de error

Código	Significado	Uso en API REST
400 Bad Request	Error del cliente	Datos enviados incorrectos o inválidos.
401 Unauthorized	No autenticado	El usuario no ha iniciado sesión.
403 Forbidden	Acceso denegado	El usuario no tiene permisos para acceder al recurso.
404 Not Found	No encontrado	El recurso no existe.
409 Conflict	Conflicto	Error debido a datos duplicados o reglas de negocio.
422 Unprocessable Content	Entidad no procesable	Error de validación de datos (usado por Pydantic).

Errores de ejecución

Código	Significado	Uso en API REST
500 Internal Server Error	Error del servidor	Fallo inesperado en el servidor.

Estos errores surgen si hay errores (**excepciones**) en tiempo de ejecución.

Deben ser **capturados y gestionados** para evitar que la aplicación falle.

Controladores

- Los **controladores** son funciones que gestionan las peticiones HTTP.
- Reciben y validan los datos de entrada
- Llaman a los servicios.
- Devuelven una respuesta al cliente.

Controladores

- Reciben la sesión mediante **inyección de dependencias**.
- Esto permite realizar todas las operaciones dentro de una **transacción**.

```
1 app = FastAPI()  
2  
3 @app.get("/users", response_model=list[UserListResponseModel])  
4 def get_users(db : Session = Depends(get_session)) -> list[UserListResponse  
5     users = user_service.get_all(db)  
6     return users
```

Exponer sólo lo necesario

Los endpoints no deben exponer más información de la necesaria.

```
1 @app.get("/users/{id}")
2 def get_user_by_id(id: int):
3     user = user_service.get_by_id(id)
4     return user
```

Exponer sólo lo necesario

Al devolver modelos Pydantic se filtrarán automáticamente los campos a devolver:

```
1 class UserResponseModel(BaseModel):
2     id: int
3     username: str
4
5 @app.get("/users/{id}", response_model=UserResponseModel)
6 def get_user_by_id(id: int):
7     user = user_service.get_by_id(id)
8     return user
```

Devolver datos agregados

Los modelos Pydantic también nos permiten devolver información agregada de varios modelos:

```
1 class PostResponseModel(BaseModel):
2     username: str
3     text: str
4
5 @app.get("/posts/{id}", response_model=PostResponseModel)
6 def get_post_by_id(id: int):
7     post = post_service.get_by_id(id)
8     return PostResponseModel(username=post.user.username, text=post.text)
```


Prevención de ataques

- Los datos de entrada deben ser validados para evitar errores y ataques (inyección SQL, XSS, ...).
- Mantener la aplicación segura es fundamental para **proteger los datos y la privacidad de los usuarios.**

Validación de datos de entrada

Ejemplo de validación manual:

```
1 @app.post("/users/{name}")
2 async def get_user_by_name(name: str):
3     if not name:
4         raise HTTPException(status_code=400, detail="Name cannot be empty")
5     if len(name) > 1000:
6         raise HTTPException(status_code=400, detail="Name is too long")
7     if any(bad_word in name.lower() for bad_word in [
8         "<script>",
9         "drop table",
10        "select *",
11        "delete from",
12        "update users",
13        "password",
14        "' or 1=1",
15        "http://",
16        "https://"
17    ]):
18        raise HTTPException(status_code=400, detail="Name contains invalid
```

Validación de datos de entrada

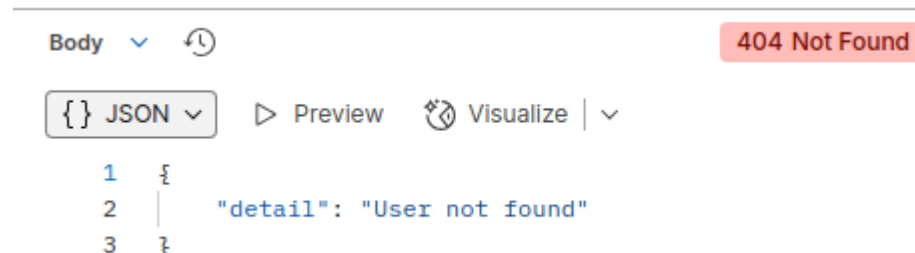
`SQLModel` y `Pydantic` permiten realizar la validación automáticamente.

Estas librerías son muy útiles para **evitar errores y proteger la aplicación**.

Controladores

Para devolver errores lanzamos `HTTPException`.

```
1 from fastapi import FastAPI, HTTPException
2
3 app = FastAPI()
4
5 @app.get("/users/{id}", response_model=UserResponseModel)
6 def get_user(id: int, db : Session = Depends(get_session)) -> UserResponseM
7     try:
8         user = user_service.get_by_id(id, db)
9         return user
10    except ValueError as e:
11        raise HTTPException(status_code=404, detail=str(e))
```



Routers

- A medida que la aplicación crece, es útil organizar los controladores en **routers**.
- Los routers agrupan los controladores por funcionalidad (usuarios, posts, etc.).
- Se pueden **montar en la aplicación principal**.

```
1 from controllers import user_router, post_router
2
3 app.include_router(user_router, tags=["users"])
4 app.include_router(movie_router, tags=["movies"])
```

Routers

controllers/user_controller.py

```
1 from fastapi import APIRouter
2
3 router = APIRouter()
4
5 @router.get("/users", response_model=list[UserListResponseModel])
6 def get_users(db : Session = Depends(get_session)) -> list[UserListResponse
7     users = user_service.get_all(db)
8     return users
```

controllers/__init__.py

```
1 from .user_controller import router as user_router
2 from .post_controller import router as post_router
```

Seguridad

Seguridad

- Las aplicaciones deben protegerse contra ataques y accesos no autorizados.
- Se pueden utilizar distintas técnicas de seguridad:
 - **Autenticación y autorización:** Proteger los endpoints con credenciales.
 - **Cifrado:** Proteger los datos en tránsito y en reposo.

Autenticación y autorización

- **Autenticación:** Verificar la identidad de un usuario.
- **Autorización:** Determinar si un usuario tiene permiso para acceder a un recurso.

Autenticación y autorización

Métodos comunes de autenticación:

- **API KEY:** Clave de acceso para proteger los endpoints.
- **Tokens JWT:** Tokens de acceso con información del usuario.
- **OAuth:** Protocolo de autorización para aplicaciones.

Una vez el usuario se ha autenticado, la aplicación puede comprobar si tiene permisos para acceder a un recurso.


Tokens JWT

JSON Web Token (JWT)

- Cadena de texto que contiene información del usuario y una firma.
 - **Payload:** Datos del usuario codificados en Base64.
 - **Firma:** Verifica la autenticidad del token.
- Cuando el usuario introduce sus credenciales (usuario y contraseña), el servidor genera y envía un token al cliente.


Tokens JWT

JWT Token

145 chars  Copy

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiaWxpY2UiLCJleHBpcmVzIjoxNzQxNjIzNDQxLjQxMTM2NDh9.CI3dE2Q2jvt6Vh2cK7dGMGVQ0kQ5-kavTJ_SjfIdH0U
```

Decoded JWT

 Copy

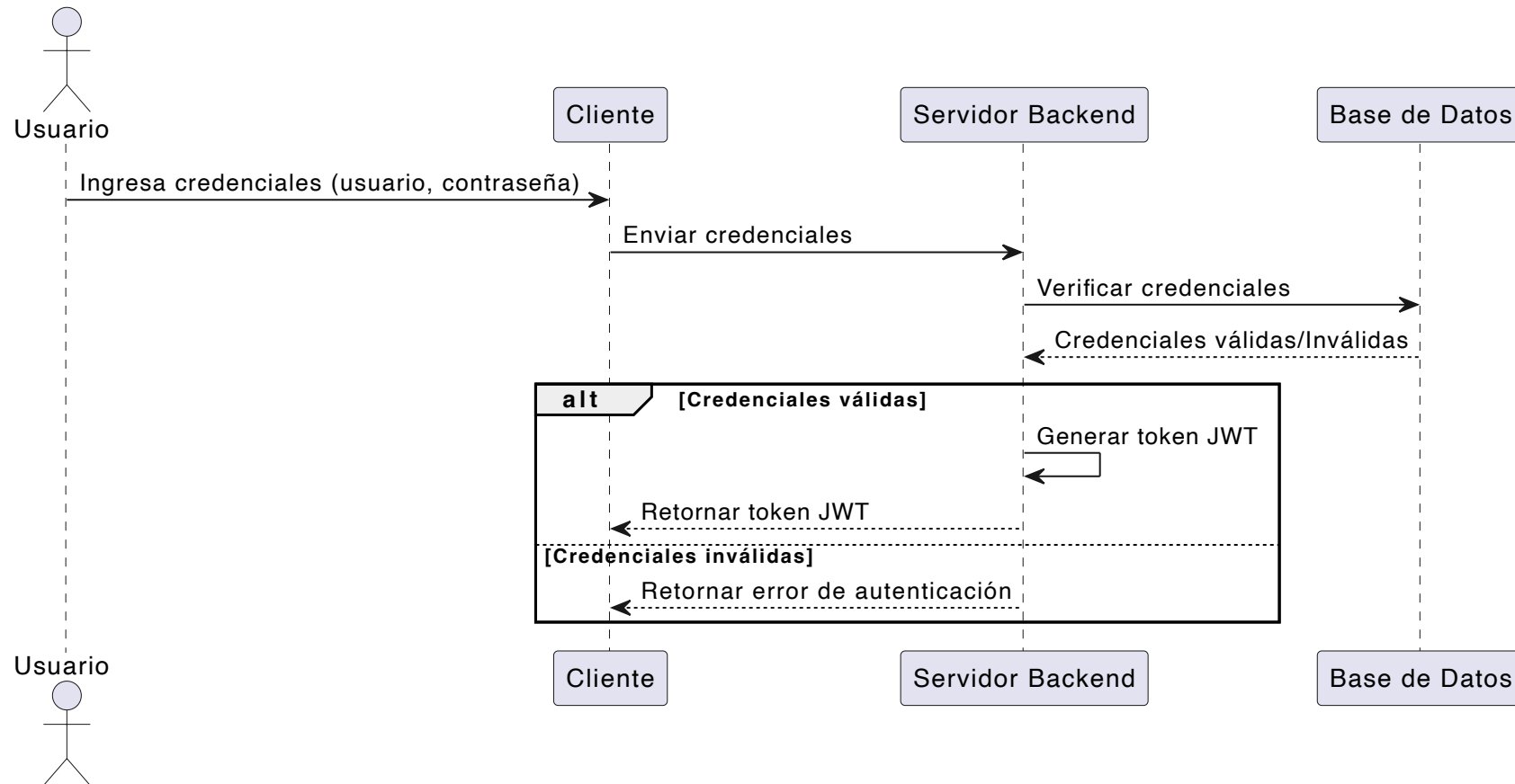
```
{ 2 items
  "header": { 2 items
    "alg": "HS256"
    "typ": "JWT"
  }
  "payload": { 2 items
    "user_id": "alice"
    "expires": 1741623441.4113648
  }
}
```

Explanation

Field	Value	Explanation
alg	HS256	the algorithm used for signing the JWT
typ	JWT	always set to "JWT"
user_id	alice	
expires	1741623441.4113648	

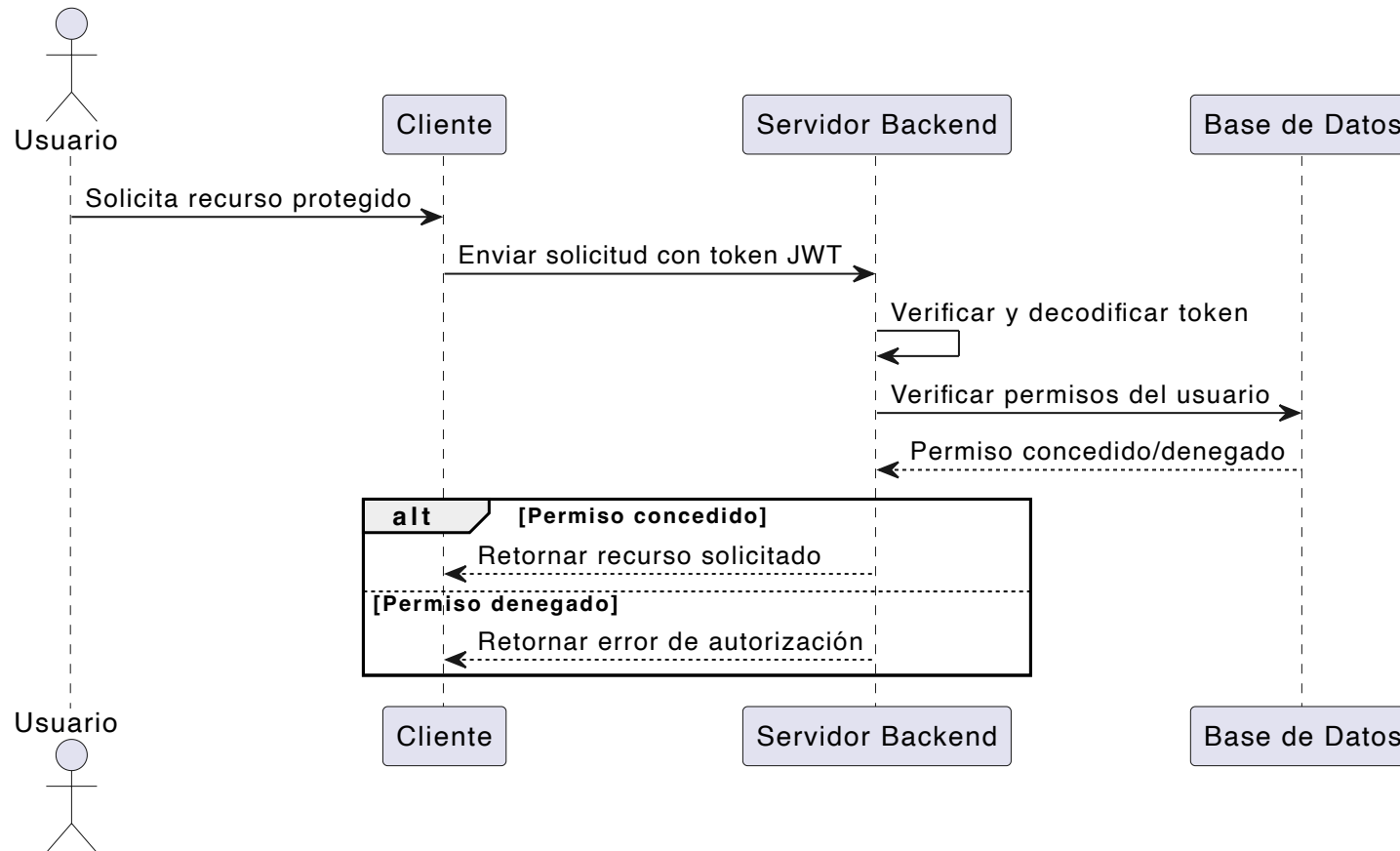
Tokens JWT

Solicitud del token (autenticación)



Tokens JWT

Solicitud del token (autorización)



Tokens JWT

El cliente debe guardar el token y enviarlo en cada petición.

- **Ventajas:** Fácil de implementar, escalable y seguro.
- **Desventajas:** Si el token es robado, el atacante puede acceder a la cuenta del usuario.

Para aumentar la seguridad se pueden añadir mecanismos de **refresco y revocación de tokens**.

Cifrado

- **Cifrado en tránsito:** Proteger los datos que se envían entre el cliente y el servidor.
 - **HTTPS:** Protocolo seguro que cifra los datos.
- **Cifrado en reposo:** Proteger los datos almacenados en la base de datos.
 - **Encriptación:** Cifrar los datos antes de almacenarlos.
 - **Hashing:** Almacenar contraseñas como hash.