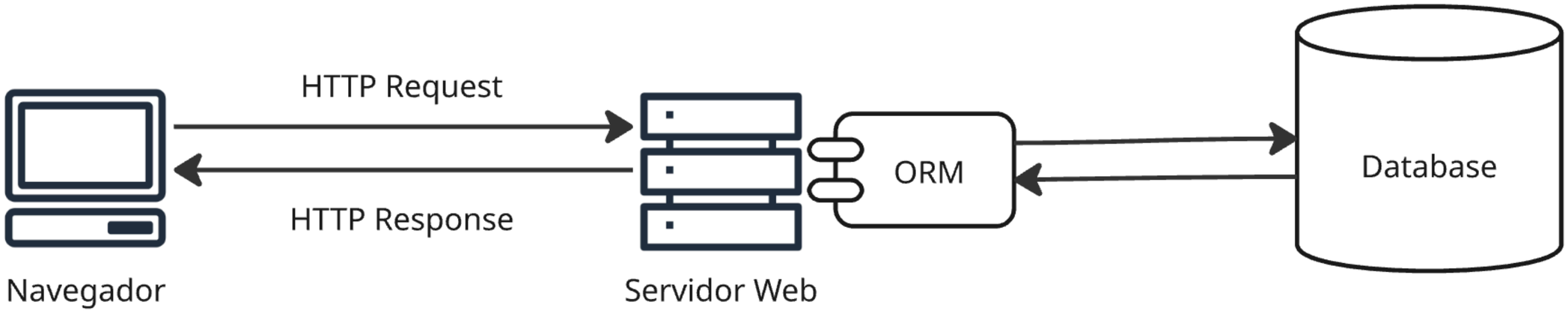
ORM

Persistencia en base de datos

ORM

- Los ORM son librerías que se encargan de servir como intermediarios en la comunicación que tendrá el backend con la base de datos.
- Beneficios:
  - Se utiliza un mismo lenguaje para la persistencia de datos (no lenguaje SQL).
  - Permite el cambio de motor de base de datos (Postgresql, Oracle, MySQL, etc).

Arquitectura de comunicación



Conceptos Generales

- **RDBMS (Relational DataBase Management System):** Motor que se encarga de procesar las consultas (**DDL, DML**) que se realizan a una base de datos.
- **Cliente de conexión:** Aplicativo que te permite realizar consultas a un RDBMS, como por ejemplo Oracle Instant Client, pgadmin 4, MySQL Workbench, etc.
- **Base de datos:** Es un repositorio donde se almacenan datos. Es servido por un **RDBMS**.

Postgresql

- Postgres es un servidor de base de datos robusto, diseñado para manejar grandes volúmenes de datos y múltiples usuarios concurrentes con una fiabilidad extrema.
- Es de código abierto.
- Características:
  - Extensibilidad: Puedes crear tus propios tipos de datos, funciones y operadores.
  - Concurrencia (MVCC): Permite que múltiples usuarios lean y escriban al mismo tiempo sin bloquearse entre sí.
  - Soporte JSONB: Aunque es relacional, maneja datos NoSQL (JSON) de forma increíblemente eficiente, permitiendo indexar documentos.
  - Integridad de Datos: Es extremadamente estricto con las reglas (claves foráneas, tipos de datos), lo que evita que tu aplicación guarde "basura".

Postgresql

Cadena de conexión

- Para que un cliente pueda conectarse a una base de datos, normalmente la base de datos se identifica por una cadena de conexión, que tiene forma de URL.

postgresql://<ROLE>:<PASSWORD>@<HOST>:<PUERTO>/<DB_NAME>

postgresql://usuario:ulima123@127.0.01:5432/bd_personas

Postgresql
Instalación

- https://www.youtube.com/watch?v=4_MMY2yiOWY

```
$ pip install sqlalchemy psycopg2-binary alembic
```

Configuración con FastAPI

Archivo database.py

- Realizamos la configuración de con qué base de datos vamos a trabajar.
- Definimos propiedades de la conexión.

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"

engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

La cadena de conexión está para sqlite… ¿Cuál sería para postgres?

Configuración con FastAPI

Para postgresql

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "postgresql://usuario:ulima123@127.0.01:5432/bd_personas"

engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

Modelos

- Los modelos son mapeos de tablas de una base de datos.
- Con __tablename__ ponemos el nombre de la tabla correspondiente en la base de datos.
- Cada atributo es una columna de la tabla con las propiedades especificadas.

```
import uuid
from sqlalchemy import Column, String
from .database import Base

class User(Base):
    __tablename__ = "users"
    id = Column(
       String(36),
       primary_key=True,
       default=lambda: str(uuid.uuid4()), # Genera un UUID v4 automáticamente
       index=True )
    username = Column(String, unique=True, index=True)
    email = Column(String, unique=True)
```

Conectar SQLAlchemy con FastAPI

Creando schema y función de apertura/cierre de conexión

```
from pydantic import BaseModel
from uuid import UUID

class UserCreate(BaseModel):
    email : str
    username : str

class User(BaseModel):
    id: UUID
    email: str
    username: str

    class Config:
        # Pydantic v2: permite leer datos que no son diccionarios (como objetos de SQLAlchemy)
        from_attributes = True
```

```
# En main.py o database.py
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Operaciones con SQLAlchemy

Crear un nuevo registro

- Abriremos una conexión a bd por cada petición HTTP que llegue a un endpoint.
- Al terminar la petición, se cerrará la conexión a la bd.

```
from fastapi import FastAPI, Depends
from sqlalchemy.orm import Session
from . import models, schemas, database

app = FastAPI()

@app.post("/users/")
def create_user(user: schemas.UserCreate, db: Session = Depends(database.get_db)):
    db_user = models.User(username=user.username, email=user.email)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user
```

Operaciones con SQLAlchemy

Leer registros

```
# 2. READ (All): Listar usuarios con paginación
@router.get("/users", response_model=List[schemas.User])
def read_users(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    usuarios = db.query(models.User).offset(skip).limit(limit).all()
    return usuarios

# 3. READ (One): Obtener un usuario por su ID
@router.get(“/users/{user_id}", response_model=schemas.User)
def read_user(user_id: int, db: Session = Depends(get_db)):
    usuario = db.query(models.User).filter(models.User.id == user_id).first()
    if not usuario:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")
    return usuario
```

Operaciones con SQLAlchemy

Actualizar

```
# 4. UPDATE: Actualizar datos de un usuario
@router.put(“/users/{user_id}", response_model=schemas.User)
def update_user(user_id: UUID, user_update: schemas.UserUpdate, db: Session = Depends(get_db)):
    user_query = db.query(models.User).filter(models.User.id == user_id)
    usuario_existente = user_query.first()

    if not usuario_existente:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")

    # Convertimos el esquema a diccionario excluyendo los campos no enviados
    update_data = user_update.dict(exclude_unset=True)
    user_query.update(update_data)

    db.commit()
    db.refresh(usuario_existente)
    return usuario_existente
```

Operaciones con SQLAlchemy

Eliminar

```
# 5. DELETE: Eliminar un usuario
@router.delete(“/users/{user_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_user(user_id: UUID, db: Session = Depends(get_db)):
    usuario = db.query(models.User).filter(models.User.id == user_id).first()

    if not usuario:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")

    db.delete(usuario)
    db.commit()
    return None
```

Migraciones

Migraciones

- Hasta el momento se necesitan crear las tablas directamente en el RDBMS, esto es una entorno distinto a la aplicación.
- Con las migraciones, las sentencias DDL serán ejecutadas desde sentencias Python (alembic) y la sincronización será automática.
- Nos permite guardar un histórico de cambios, similar a lo que tenemos con GIT con el código.

Alembic

Iniciar proyecto

- Debemos configurar alembic para que utilice nuestra base de datos y nuestros modelos.

```
$ pip install alembic
$ alembic init alembic
```

```
# En env.py
from app.database import Base  # Importar tu Base
from app.models import user    # Importar tus modelos
target_metadata = Base.metadata
```