Setting up a FastAPI App with Async SQLALchemy 2.0 & Pydantic V2

Thomas Aitken · Follow
11 min read · Oct 20

Early this year, a major update was made to SQLAlchemy with the release of SQLAlchemy 2.0. Among other things, this includes significant updates to basic ORM syntax and to some technical machinery necessary for a good experience using an async engine with *asyncio*. (You can read more about these updates here: https://docs.sqlalchemy.org/en/20/changelog/whatsnew_20.html.)

Improvements to the async experience are a boon for the use of SQLAlchemy with FastAPI in particular, as having an async database sesson means that you no longer have to worry about database operations blocking async path operations. In short, it offers a big speed-up over a synchronous database setup.

I have long been attracted to the innovativeness and elegance of FastAPI — the way it uses dependencies to create a 'functional' design, the way it

cleverly marries with other open source libraries like Pydantic, and its async-first architecture. But until my latest web app project, I had never used it before.

It turned out that there was a lot to learn! First, learning the latest patterns on the FastAPI side; next, working out how to use it with the latest versions of SQLAlchemy and Pydantic; and, finally, how to set up a robust infrastructure around it (migrations, tests, etc). Because of how fast-moving the ecosystem is (there have been recent major upgrades to SQLAlchemy and Pydantic, and FastAPI is constantly moving), there is not a whole lot of documentation out there about how to do things with all the latest versions. Moreover, unlike Django, FastAPI does not give you much out of the box. That's why I'm going to save you the effort I went through!

Without further ado, here is a guide to setting up your FastAPI project using SQLAlchemy 2.0+, Pydantic 2.0+, Alembic 1.11, Pytest and FastAPI 0.100+ with Python 3.11.

> *Who is this guide for?*
>
> This guide is pitched at Python developers of an intermediate to advanced level who have at least some familiarity with web app development. It would be most useful to those who want to start a FastAPI project from scratch using all the latest tools and techniques. Many things will be assumed knowledge, including Docker containers, Python virtual environments, database models, database migrations, concurrency in Python, and Python type annotations. If you don't know what some of these things are, you should start somewhere else, e.g. with the main FastAPI docs.

Note: all the code for this guide can be found here: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/. This is

not a complete set up but merely a skeleton for you to build on in your own project.

Installation

I am accustomed to using Docker for local development, so I started this project using Docker Compose. If you don't want to use Docker Compose for your setup, it should still be possible to replicate this installation in a non-dockerized fashion. However, I will not give an alternate set of instructions since those can be found elsewhere.

Here is my `docker-compose.yml` for development:

```
version: "3.7"
services:
  postgres:
    image: postgres:15-alpine
    restart: always
    environment:
      POSTGRES_USER: dev-user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: dev_db
    ports:
      - 5432:5432
    expose:
      - "5432"
    volumes:
      - db-data:/var/lib/postgresql/data:cached

  test-postgres:
    image: postgres:15-alpine
    restart: always
    environment:
      POSTGRES_USER: test-user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: test_db
    ports:
      - 5434:5432 # Use a different port to avoid conflicts with the main databa
    expose:
      - "5434" # Exposing the different port for clarity
    volumes:
      - test-db-data:/var/lib/postgresql/data:cached

  backend:
    build:
```

```
      context: backend
      dockerfile: Dockerfile
    command: python app/main.py
    tty: true
    volumes:
      - ./backend:/backend/:cached
      - ./.docker/.ipython:/root/.ipython:cached
    environment:
      PYTHONPATH: .
      DATABASE_URL: "postgresql+asyncpg://dev-user:password@postgres:5432/dev_db
    depends_on:
      - "postgres"
    ports:
      - 8000:8000

  frontend:
    build:
      context: frontend
      dockerfile: Dockerfile
    stdin_open: true
    volumes:
      - "./frontend:/app:cached"
      - "./frontend/node_modules:/app/node_modules:cached"
    environment:
      - NODE_ENV=development
    ports:
      - 3000:3000

volumes:
  db-data:
  test-db-data:
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/docker-compose.yml

(Note: I set up my FastAPI app as a backend for a React app so that is why you see a `frontend` container.)

The main things you should see are that I am using Postgres for my DB and `asyncpg` as my database interface library, and that I have set up two persistent DBs for local development: one for running the app and another for testing.

Why dockerize your local DB instances?

This makes it easy for new developers to spin up this project without having to install a bunch of complicated libraries.

Why use a persistent DB for testing?

As we'll cover later, the setup I'm using for Pytest ensures that no actual data persists to this DB from tests, but making the volume persistent does save us from running every single migration every time we want to run tests.

Meanwhile, here is the `Dockerfile` I'm using for the backend container:

```
FROM python:3.11.4

RUN mkdir /backend
WORKDIR /backend

RUN apt update && \
    apt install -y postgresql-client

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

COPY . .
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/Dockerfile

So far, so simple. Let's move on.

Models and Schemas

In FastAPI the convention is to separate out your SQLAlchemy model classes from your Pydantic schemas — the SQLAlchemy classes are used only for

defining the DB schema, the schemas are for validating incoming and outgoing data in your crud functions and path operations. (Pydantic schemas are sort of half-way between API serializer classes and dataclasses — which is a very handy hybrid to have.)

Now, it's true that the creator of FastAPI has created a project, SQLModel, that tries to unify SQLAlchemy models and Pydantic schemas into one entity. But this library seems to be a little behind the curve in terms of keeping up with the latest developments in either SQLAlchemy or Pydantic. In any case, I prefer the pattern of separating the DB classes from the schemas. One reason to want this is to have different schemas for different CRUD operations — e.g. your schema for updates may have more optional types than your schema for reading.

Ok, so with that in mind, how do we define our DB classes? Coming from a Django background, I am used to storing my model classes in a folder structure like the following:

```
models/
    __init__.py
    model_1.py
    model_2.py
    ...
```

Naturally, I tried to do the same thing using the latest way of defining model classes in SQLAlchemy 2.0, i.e.Mapped Classes. I was able to get this to work, but there's a trick. If you want to expose your models in the `__init__.py` of this directory, you will need to define the required base class before you load in the models themselves. In other words, you will need to do something like this:

```
# app/models/__init__.py

from sqlalchemy.orm import declarative_base

Base = declarative_base() # model base class

from .model_1 import Model1
from .model_2 import Model2
```

From here you can define your first model class using the new SQLAlchemy pattern. Here's the example from the demo project:

```
from sqlalchemy.orm import Mapped, mapped_column

from . import Base


class User(Base):
    __tablename__ = "user"

    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True, index=
    username: Mapped[str] = mapped_column(index=True, unique=True)
    slug: Mapped[str] = mapped_column(index=True, unique=True)
    email: Mapped[str] = mapped_column(index=True, unique=True)
    first_name: Mapped[str]
    last_name: Mapped[str]
    hashed_password: Mapped[str]
    is_superuser: Mapped[bool] = mapped_column(default=False)
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/models/user.py

Ok, now onto Pydantic schemas.

Naturally enough, we can define the schemas for this table in `/app/schemas/user.py`.

Here are some basic schemas that could be used for simple READ operations on the above `User` model:

```
from pydantic import BaseModel, ConfigDict


class User(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    id: int
    username: str
    slug: str
    email: str
    first_name: str
    last_name: str
    is_superuser: bool = False

class UserPrivate(User):
    hashed_password: str
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/schemas/user.py

Note: Pydantic's magic feature is that all of these mandatory types will be validated whenever you use this schema.

Alembic

Alembic is the tool pretty much everyone uses for managing migrations with SQLAlchemy. It requires a bit more hands-on work and attention than Django's migrations library but it's more explicit and decently reliable.

Alembic requires some tweaking to work with an async SQLAlchemy engine. The main file for customising your Alembic set up is `alembic/env.py`. Here's the one from the demo project:

```
import asyncio
import os
from logging.config import fileConfig

from alembic import context
from app.models import Base
from asyncpg import Connection
from sqlalchemy import pool
from sqlalchemy.ext.asyncio import async_engine_from_config

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

# Interpret the config file for Python logging.
# This line sets up loggers basically.
fileConfig(config.config_file_name)  # type: ignore

# add your model's MetaData object here
target_metadata = Base.metadata


def get_url():
    return os.getenv("DATABASE_URL")


def run_migrations_offline():
    """Run migrations in 'offline' mode.
    This configures the context with just a URL
    and not an Engine, though an Engine is acceptable
    here as well.  By skipping the Engine creation
    we don't even need a DBAPI to be available.
    Calls to context.execute() here emit the given string to the
    script output.
    """
    # url = config.get_main_option("sqlalchemy.url")
    url = get_url()
    context.configure(
        url=url,
        target_metadata=target_metadata,
        literal_binds=True,
        dialect_opts={"paramstyle": "named"},
    )

    with context.begin_transaction():
        context.run_migrations()


def do_run_migrations(connection: Connection) -> None:
    context.configure(connection=connection, target_metadata=target_metadata)

    with context.begin_transaction():
```

```
        context.run_migrations()


async def run_migrations_online():
    """Run migrations in 'online' mode.
    In this scenario we need to create an Engine
    and associate a connection with the context.
    """
    configuration = config.get_section(config.config_ini_section)
    configuration["sqlalchemy.url"] = get_url()
    connectable = async_engine_from_config(
        configuration,
        prefix="sqlalchemy.",
        poolclass=pool.NullPool,
    )

    async with connectable.connect() as connection:
        await connection.run_sync(do_run_migrations)

    await connectable.dispose()


if context.is_offline_mode():
    run_migrations_offline()
else:
    asyncio.run(run_migrations_online())
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/alembic/env.py

From here you can autogenerate your first migration file like so:

```
docker compose exec backend alembic revision --autogenerate -m "your message her
```

And run this migration like so:

```
docker compose exec backend alembic upgrade head
```

App Settings

There's a handy spinoff of Pydantic called `pydantic-settings` that provides a typesafe settings class that automatically loads values from env variables. Here's a simple example:

```
from pydantic_settings import BaseSettings


class Settings(BaseSettings):
    database_url: str
    echo_sql: bool = True
    test: bool = False
    project_name: str = "My FastAPI project"
    oauth_token_secret: str = "my_dev_secret"


settings = Settings()  # type: ignore
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/config.py

To give an example of how this works: if you set an environment variable DATABASE_URL = "blah" then you will be able to instantly access that value in your app by importing this class instance. This is very handy for managing values in different environments.

Database Session

Now we get to the really interesting part, how to set up your async engine and async database sessions. The ideal for this is that you can use the same set up in your app and in your tests. Borrowing a lot from an excellent blog

(https://praciano.com.br/fastapi-and-async-sqlalchemy-20-with-pytest-done-right.html), here is the session manager class I am using:

```
import contextlib
from typing import Any, AsyncIterator
```



```
    async_sessionmaker,
    create_async_engine,
)
from sqlalchemy.orm import declarative_base

Base = declarative_base()

# Heavily inspired by https://praciano.com.br/fastapi-and-async-sqlalchemy-20-wi


class DatabaseSessionManager:
    def __init__(self, host: str, engine_kwargs: dict[str, Any] = {}):
        self._engine = create_async_engine(host, **engine_kwargs)
        self._sessionmaker = async_sessionmaker(autocommit=False, bind=self._eng

    async def close(self):
        if self._engine is None:
            raise Exception("DatabaseSessionManager is not initialized")
        await self._engine.dispose()

        self._engine = None
        self._sessionmaker = None

    @contextlib.asynccontextmanager
    async def connect(self) -> AsyncIterator[AsyncConnection]:
        if self._engine is None:
            raise Exception("DatabaseSessionManager is not initialized")

        async with self._engine.begin() as connection:
            try:
                yield connection
            except Exception:
                await connection.rollback()
                raise

    @contextlib.asynccontextmanager
    async def session(self) -> AsyncIterator[AsyncSession]:
        if self._sessionmaker is None:
            raise Exception("DatabaseSessionManager is not initialized")
```

```
        session = self._sessionmaker()
        try:
            yield session
        except Exception:
            await session.rollback()
            raise
        finally:
            await session.close()


sessionmanager = DatabaseSessionManager(settings.database_url, {"echo": settings


async def get_db_session():
    async with sessionmanager.session() as session:
        yield session
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/database.py

At this point I may as well finally show you the main file. This looks as follows:

```
import logging
import sys
from contextlib import asynccontextmanager

import uvicorn
from app.api.routers.users import router as users_router
from app.config import settings
from app.database import sessionmanager
from fastapi import FastAPI

logging.basicConfig(stream=sys.stdout, level=logging.DEBUG if settings.debug_log


@asynccontextmanager
async def lifespan(app: FastAPI):
    """
    Function that handles startup and shutdown events.
    To understand more, read https://fastapi.tiangolo.com/advanced/events/
    """
    yield
```

```
    if sessionmanager._engine is not None:
        # Close the DB connection
        await sessionmanager.close()


app = FastAPI(lifespan=lifespan, title=settings.project_name, docs_url="/api/doc


@app.get("/")
async def root():
    return {"message": "Hello World"}


# Routers
app.include_router(users_router)


if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", reload=True, port=8000)
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/main.py

Notice the `lifespan` context manager which will close the DB connection automatically when the app is closed.

Now, recall the `get_db_session` function from the `database.py` file shared above. This can become our fundamental db interface across the app. First, we turn it into a dependency:

```
from typing import Annotated

from app.database import get_db_session
from fastapi import Depends
from sqlalchemy.ext.asyncio import AsyncSession

DBSessionDep = Annotated[AsyncSession, Depends(get_db_session)]
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/api/dependencies/core.py

Then we can use it like so:

```
from app.api.dependencies.auth import validate_is_authenticated
from app.api.dependencies.core import DBSessionDep
from app.crud.user import get_user
from app.schemas.user import User
from fastapi import APIRouter, Depends

router = APIRouter(
    prefix="/api/users",
    tags=["users"],
    responses={404: {"description": "Not found"}},
)

@router.get(
    "/{user_id}",
    response_model=User,
    dependencies=[Depends(validate_is_authenticated)],
)
async def user_details(
    user_id: int,
    db_session: DBSessionDep,
):
    """
    Get any user details
    """
    user = await get_user(db_session, user_id)
    return user
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/api/routers/users.py

CRUD

With a database session in hand, we can begin to define CRUD functions like the `get_user` function defined in the previous snippet. Using the latest SQLAlchemy syntax, the implementation looks like this:

```
from app.models import User as UserDBModel
from fastapi import HTTPException
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession


async def get_user(db_session: AsyncSession, user_id: int) -> UserDBModel:
    user = (await db_session.scalars(select(UserDBModel).where(UserDBModel.id ==
    if not user:
        raise HTTPException(status_code=404, detail="User not found")
    return user
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/crud/user.py

N.B.: if you are writing a Create or Update function, I would recommend not calling `await db_session.commit()` inside the function itself — instead do it in the path operation function. This is for two reasons:

1. You will sometimes want to compose multiple CRUD functions and usually you will want them to occur in the same session (transaction).
2. It makes test cleanup way easier (next section).

Async Testing with Pytest

Pytest is an insanely powerful and versatile Python testing framework for unittests and integration tests. The main configuration file that Pytest uses is called `conftest.py`. In this file, you can define the "fixtures" (basically, pre-computed functions) that you need for your tests. These fixtures are highly controllable: they can be run before every test, or at the start of the entire test session; they can be run in the background, or you can use the return value as an argument to your test functions.

The requirements for my `conftest.py` were as follows:

1. At the start of the session, it should automatically connect to the test DB and run migrations if any are missing.
2. Each test function should be a clean slate — nothing should be written to the test DB.
3. We should be able to use `db_session` as a test argument or have it set up automatically in the background.

With much help from ChatGPT, and a lot of tweaking, I found that this file achieved these objectives:

```
import asyncio
from contextlib import ExitStack

import pytest
from alembic.config import Config
from alembic.migration import MigrationContext
from alembic.operations import Operations
from alembic.script import ScriptDirectory
from app.config import settings
from app.database import Base, get_db_session, sessionmanager
from app.main import app as actual_app
from asyncpg import Connection
from fastapi.testclient import TestClient


@pytest.fixture(autouse=True)
def app():
    with ExitStack():
        yield actual_app


@pytest.fixture
def client(app):
    with TestClient(app) as c:
        yield c


@pytest.fixture(scope="session")
def event_loop(request):
    loop = asyncio.get_event_loop_policy().new_event_loop()
    yield loop
    loop.close()
```

```python
def run_migrations(connection: Connection):
    config = Config("app/alembic.ini")
    config.set_main_option("script_location", "app/alembic")
    config.set_main_option("sqlalchemy.url", settings.database_url)
    script = ScriptDirectory.from_config(config)

    def upgrade(rev, context):
        return script._upgrade_revs("head", rev)

    context = MigrationContext.configure(connection, opts={"target_metadata": Ba

    with context.begin_transaction():
        with Operations.context(context):
            context.run_migrations()


@pytest.fixture(scope="session", autouse=True)
async def setup_database():
    # Run alembic migrations on test DB
    async with sessionmanager.connect() as connection:
        await connection.run_sync(run_migrations)

    yield

    # Teardown
    await sessionmanager.close()


# Each test function is a clean slate
@pytest.fixture(scope="function", autouse=True)
async def transactional_session():
    async with sessionmanager.session() as session:
        try:
            await session.begin()
            yield session
        finally:
            await session.rollback()  # Rolls back the outer transaction


@pytest.fixture(scope="function")
async def db_session(transactional_session):
    yield transactional_session


@pytest.fixture(scope="function", autouse=True)
async def session_override(app, db_session):
    async def get_db_session_override():
        yield db_session[0]

    app.dependency_overrides[get_db_session] = get_db_session_override
```

Source: https://github.com/ThomasAitken/demo-fastapi-async-sqlalchemy/blob/main/backend/app/conftest.py

The only detail missing here is we need to set `settings.database_url` to be the test database. The way I am doing this is when executing the test command itself:

```
TEST=1 DATABASE_URL="postgresql+asyncpg://test-user:password@test-postgres:5432/
```

With this `confest.py` in place, define your `pytest.ini` to handle async test functions as follows:

```
[pytest]
asyncio_mode = auto
```

Then, voila, you can define a test like so:

```
async def test_my_crud_function(db_session: AsyncSession):
    pass
```

Note: you want to avoid calling `await db_session.commit()` in these test functions because that will write to the test DB. If you must, you will need to follow the latest "savepoint" (nested transaction) docs.

The End

I admit, that was a lot. If you've got this far, you've done well and I salute you . But hey, nobody ever said using FastAPI for a web app project was easy, at least not compared to Django.

I just hope this guide will make setting up your own project a bit less daunting.

Fastapi Sqlalchemy Pydantic Python Pytest

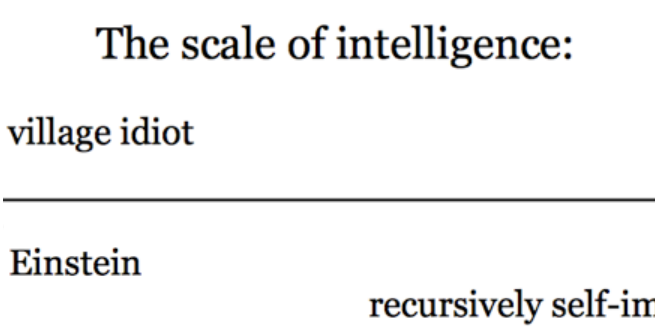
Written by Thomas Aitken

Follow

13 Followers

A person with wide-ranging intellectual interests who enjoys reading non-fiction books. I write long essays.

More from Thomas Aitken



Thomas Aitken

A Different Critique of the Fear of Superintelligence

Pretty much anyone involved in circles that intersect with the technology industry or...

21 min read · Mar 25, 2022

2

Thomas Aitken

Degrowth or Transhumanism: Humanity's Forced Choice

The modern neologisms "Degrowth" and "Transhumanism" refer to two starkly...

14 min read · Aug 25

1

Thomas Aitken

The Shortest Argument Against A Superintelligence Explosion

Here's a standard, quasi-syllogistic statement of the argument for the plausibility of a...

3 min read · Jun 1

100

See all from Thomas Aitken

Recommended from Medium

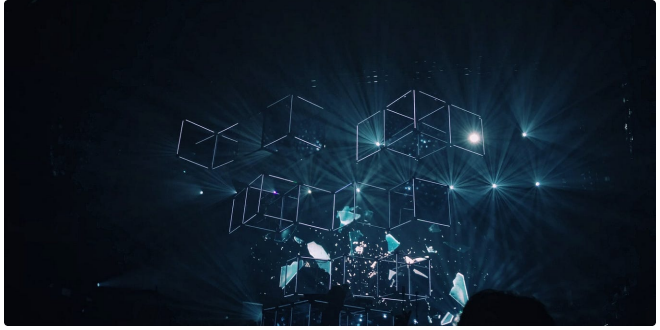
Itay

Beyond FastAPI: The Evolution of Python Microservices in 2024 wit...

TL;DR: PyNest emerges as the superior framework for building APIs and...

3 min read · Nov 9

392 7



Anand Rathore in AWS Tip

Data Class in Python

Unlock Code Clarity with Python's @dataclass: Simplify Data Management,...

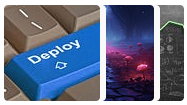
9 min read · Nov 15

45

Lists

Coding & Development

11 stories · 289 saves

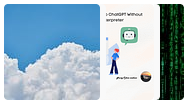


Predictive Modeling w/ Python

20 stories · 643 saves

Practical Guides to Machine Learning

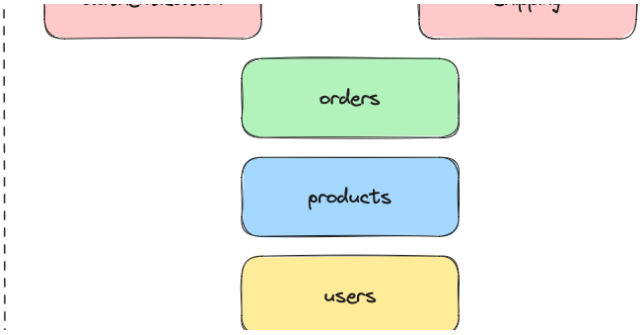
10 stories · 723 saves



ChatGPT

22 stories · 282 saves

Ayush Thakur



Dmytro Parfeniuk in Python in Plain English

Wait! What are Pipelines in Python?

If you are a Python developer, you might have heard of the term pipeline. But what exactly i...

5 min read · Nov 6

576 11

🐍 Python Backend Project Advanced Setup (FastAPI Example)

🥏 🌶️ FastAPI & Pydantic 2.4 & SQLAlchemy 2.0 & More

11 min read · Jul 23

579 3

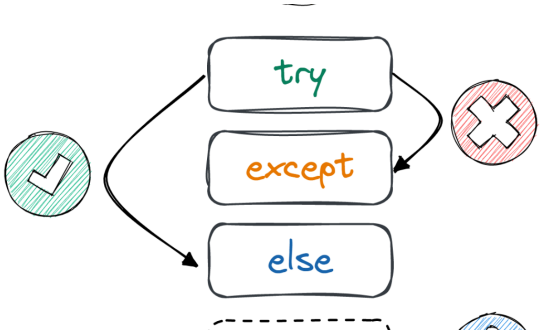
Deepak Ranolia

Python Cheatsheet

1. Data Types:

5 min read · Nov 18

89 2



Ashok Poudel in GoPenAI

Exception & Error Handling in Python for Professionals: A...

By understanding and implementing robust error handling techniques in your Python...

15 min read · Aug 4

445 4

See more recommendations