Web Development in Python for Data Teams

Strategically Unlocking Data Products Without Leaving Your Language

September 2025

DL Lim

CONTENTS

1 EXECUTIVE SUMMARY

Data teams today are tasked not only with producing accurate models and analyses but also with making their outputs accessible across the organisation. Traditional approaches often create friction: **web teams may struggle to implement complex analytical logic** in unfamiliar languages, while **data teams can be pulled away from core work** to maintain front-end interfaces.

This misalignment leads to inefficiencies, duplication of effort, and slowed delivery of data products. **By adopting Python-native web frameworks** and API-first architectures, **organisations can bridge this gap**, allowing each team to focus on their strengths while still delivering accessible, maintainable, and high-performing services.

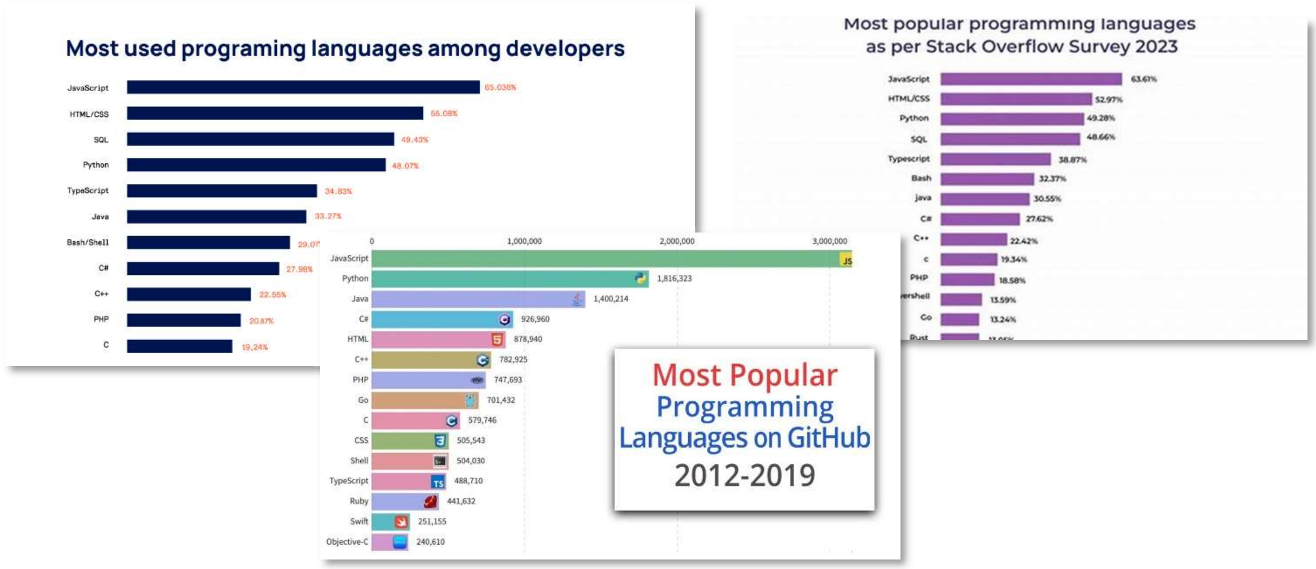
FastAPI emerges as a particularly compelling choice for data teams. Purpose-built for web APIs, it allows teams to expose analytical workflows and models with minimal overhead, while benefiting from automatic validation, documentation, and modern asynchronous capabilities. Coupled with tools like SQLModel, which unify data validation and database interaction, **data teams can manage their entire stack in Python**. This reduces handoffs, eliminates translation errors, and allows teams to version control both their data models and API schemas seamlessly.

Ultimately, these approaches enable a more scalable and strategic architecture. By building APIs that can be shared across business units or consumed by front-end applications, organisations unlock the potential of a data mesh model, promoting autonomy, consistency, and faster decision-making. In doing so, they empower data teams to deliver high-value outputs efficiently, while maintaining accuracy, reliability, and control over their analytical products.

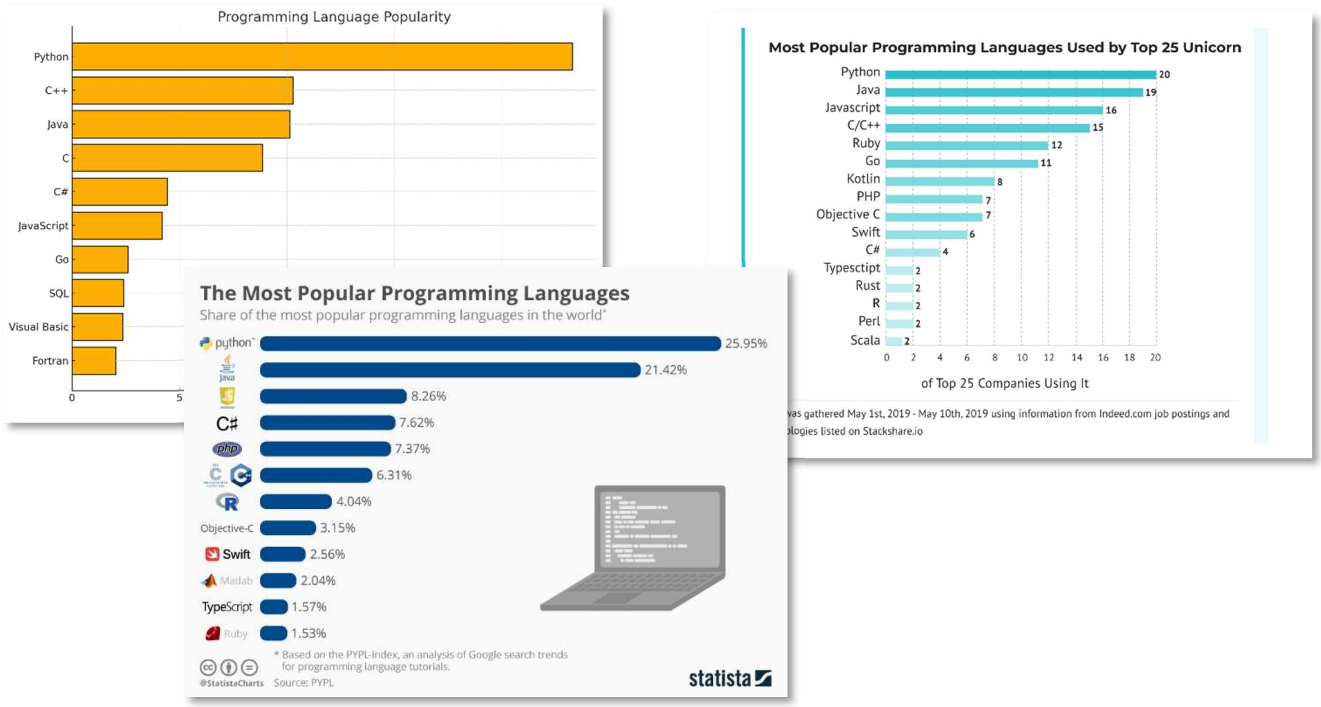
2 INTRODUCTION

Web development has long been dominated by JavaScript and its surrounding ecosystem. Front-end applications are commonly built with TypeScript frameworks such as React, Angular, or Vue, while the back end often relies on Node.js. Other languages like PHP, Ruby, Go, Java, and .NET also continue to hold their ground as popular options for powering the server side of the web.

Depending on who you ask, JavaScript is often cited as the most widely used programming language in the world.



Yet, Python stands firmly alongside it as one of the most influential languages of our time. With its clean syntax and its central role in the rise of AI, machine learning, and data science, Python has become the language of choice for data teams across industries.



3 WHY USE PYTHON

Python has earned its reputation as the language of choice for data-driven work. From data science to machine learning and artificial intelligence, it is widely considered the best tool for the job. Beyond that, it consistently ranks as one of the most versatile programming languages in use today. Colloquially second-best choice for almost everything else.

For data teams, this versatility carries practical advantages when it comes to web development:

- **One language across the stack**: A Python-based web server allows the same language to power both the analytical layer and the application backend.
- **Reduced need for specialists**: Teams no longer need to hire or train JavaScript-focused developers simply to bridge the gap between analytics and delivery.
- **Less silo-ing, fewer handoffs**: A common language reduces friction between teams, minimises knowledge gaps, and streamlines collaboration.

The result is a tech stack that is simpler to maintain and easier to evolve, with fewer moving parts and fewer dependencies on highly specialised skills. For data teams accustomed to working primarily in Python, the learning curve is minimal, enabling faster delivery of new products and features.

It should be acknowledged, however, that Python's strengths lie more in the backend than the front end. While it can support web interfaces, it is not the strongest tool for producing modern, highly polished user experiences. That challenge, and how it can be addressed, will be explored later in this paper.

4 POPULAR PYTHON WEB FRAMEWORKS

When data teams begin to consider web or API development in Python, the natural question arises: which framework should we use? Unlike the JavaScript ecosystem, which is often fragmented and rapidly evolving, the Python web development landscape is relatively stable, with a few well-established frameworks that dominate the field.

Three standouts are **Django**, **Flask**, and **FastAPI**. Each offers a different balance of simplicity, flexibility, and performance. The choice between them depends on the needs of the team, the scale of the project, and the skills of the developers involved.

The table below highlights their main characteristics and differences:

	Django	**Flask**	**FastAPI**
Philosophy	Batteries included, full-stack	Lightweight, minimal, flexible	Modern, API-first, type-driven
Primary Use Case	Large applications, full-featured platforms	Small to medium apps, microservices	APIs, microservices, async applications
Architecture	Monolithic (can be modular with effort)	Microframework, highly modular	Async-first, modular
Ease of Learning	Steeper learning curve, but very structured	Easy for beginners, minimal boilerplate	Moderate, but intuitive with type hints
Performance	Good, but slower compared to FastAPI	Good for most workloads	Excellent, among the fastest Python frameworks
Asynchronous Support	Limited (as of recent versions)	Basic via extensions, not native	Native async/await support
Built-in Features	ORM, admin panel, authentication, forms, etc.	Minimal built-in, extensions required	Auto docs (Swagger, ReDoc), dependency injection, SQLModel ORM
Extensibility	Strong ecosystem, less flexible out-of-box	Very flexible with many extensions	Good extensibility, smaller but growing ecosystem
Community & Ecosystem	Mature, very large, long history	Mature, very large, long history	Growing fast, vibrant, but younger
Documentation	Comprehensive, well established	Clear, good for beginners	Excellent, modern and developer-friendly
Deployment Complexity	More complex, but stable and well-supported	Simple, flexible, many deployment options	Straightforward, optimised for modern APIs
Best Fit For	Enterprise apps, complex data-driven systems	Prototyping, lightweight services	High-performance APIs, modern microservices

Key Takeaways:

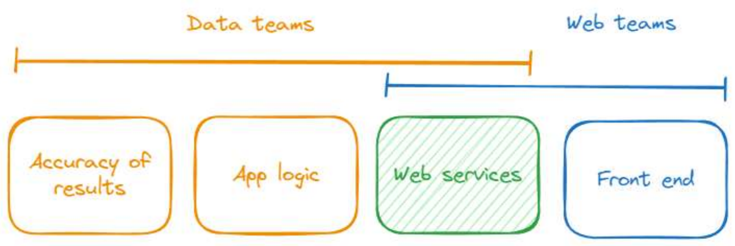
- **Django** is best suited for teams that want a **comprehensive, ready-to-go framework** where much of the architecture is predefined. It is particularly strong in enterprise-scale applications with heavy data management requirements.
- **Flask** shines in **smaller projects** or situations where maximum flexibility is required. It provides a solid foundation without imposing architectural decisions, which can be a double-edged sword.
- **FastAPI** represents the **modern wave of Python frameworks**, designed around asynchronous programming and type hints. It excels in API-first projects, offering excellent performance and built-in documentation generation.

For data teams, the right choice depends on the balance between speed of delivery, maintainability, and long-term scalability. Django is ideal if you want an all-in-one solution with minimal decision fatigue, Flask suits those who want simplicity and freedom, and FastAPI is compelling for API-driven products that demand high performance and modern practices.

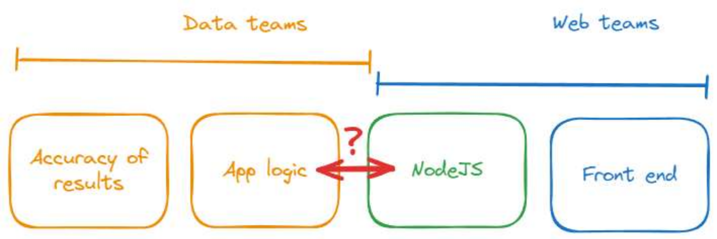
5 WHY PYTHON FOR WEB AND API DEVELOPMENT

There is an increasing need for data teams to expose insights, models, and analytical results to the web or other teams. However, questions around ownership often arise: **should data teams take responsibility for web services, or should web teams handle complex analytical logic?** This uncertainty can create gaps, inefficiencies, or duplicated effort across the organisation.

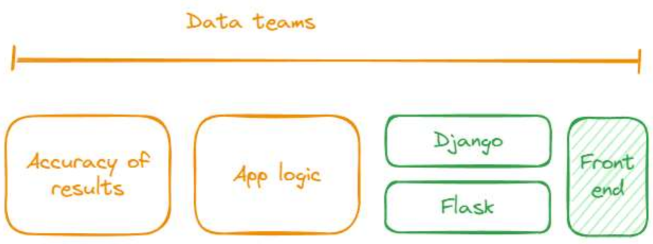
This section explores the challenges and opportunities that arise when data teams consider web development. It illustrates the tensions between data and web teams, the inefficiencies of traditional frameworks, and how API-first strategies can provide a balanced, scalable solution. Through a series of diagrams, we show how thoughtful framework choices and architectural approaches enable data teams to expose their work effectively, without losing focus on what they do best: creating accurate, impactful data products.



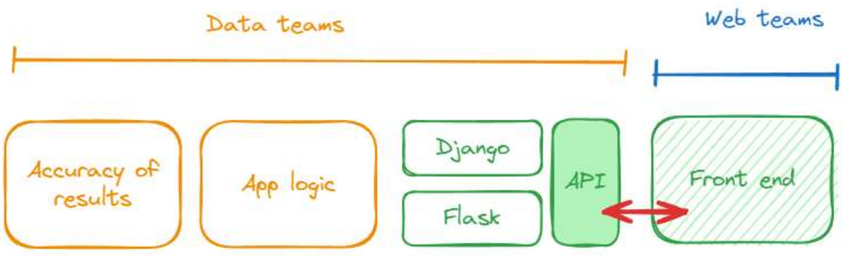
The overlap between data teams and web teams creates uncertainty around ownership of web services. **Should data teams extend into web development, or should web teams take on complex ML and analytical logic?** This ambiguity often leads to gaps, inefficiencies, or duplicated effort.



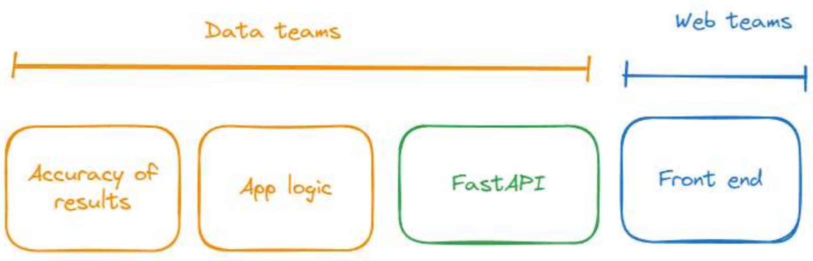
Let’s consider a situation where web teams manage back-end services in Node.js. **Complex application logic written in Python must be translated across languages**. This creates inefficiencies, introduces risks of misinterpretation, and adds friction between teams whose strengths lie in very different domains.



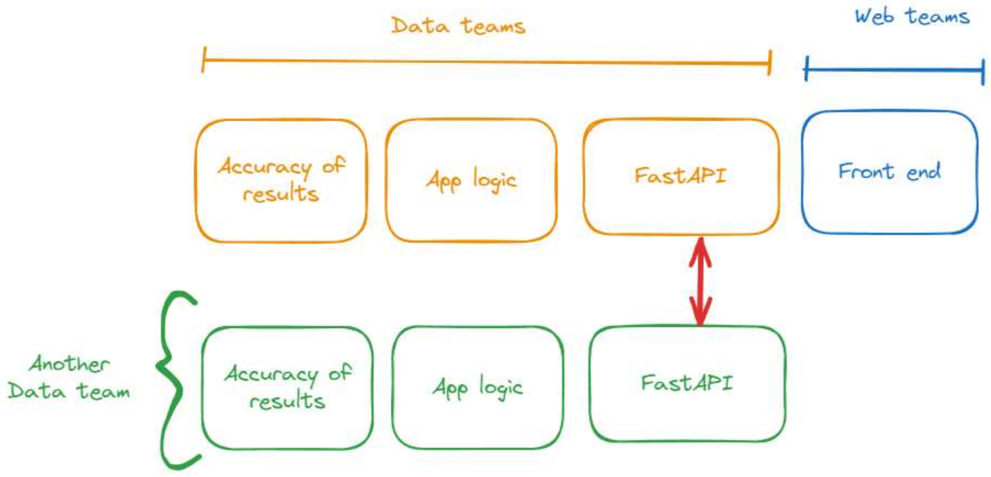
On the flip side, when data teams manage web services using frameworks like Django or Flask, they are often required to **build and maintain the front-end interfaces** themselves. This quickly becomes a full-time responsibility, diverting attention from developing core application logic and ensuring the accuracy of results. The additional burden can lead to inefficiencies and reduced focus on the data products that matter most.



A practical compromise is to **build an API that exposes the application logic**, while delegating the front-end development to the web team. This decoupling allows data teams to maintain complex application logic in Python within the API, while web teams focus full-time on delivering polished user interfaces.



Given the scenario, using FastAPI becomes a natural choice. Unlike Django or Flask, FastAPI is a modern framework purpose-built for building web APIs in Python, offering native asynchronous support, automatic documentation, and high performance tailored for data-driven applications.



When a data team builds and exposes their API to the broader organisation, it **enables a data mesh architecture.** Different data teams or business units can share and access data through these APIs, while some APIs can also be directly consumed by front-end applications, promoting scalability, autonomy, and consistency patterns across the enterprise.

By shifting the responsibility for web interfaces to specialised teams and allowing data teams to focus on API-driven delivery, organisations strike a balance between expertise, efficiency, and maintainability. This approach not only reduces duplication of effort and minimises translation errors but also lays the foundation for scalable, modern architectures where data products can be shared and consumed seamlessly across the enterprise.

6 INTRO TO FASTAPI

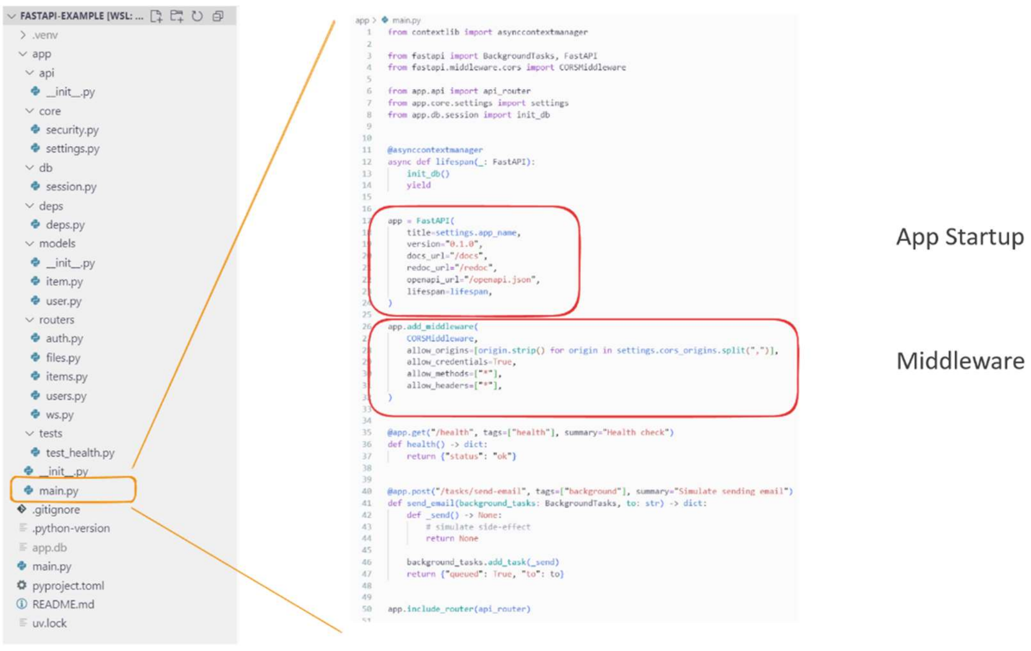
FastAPI is a modern Python framework specifically designed for building high-performance APIs. It combines simplicity, speed, and type safety, making it an ideal choice for data teams who want to expose their models and analytical workflows without leaving the Python ecosystem.

In this section, we provide a hands-on example codebase that readers can follow along with. The goal is to demonstrate FastAPI's core features, including but not limited to:

- Defining routes and endpoints with minimal boilerplate
- Automatic data validation using Python type hints
- Built-in interactive documentation with Swagger and ReDoc
- Support for asynchronous operations to handle high-performance workloads

By working through the example, data teams will gain a practical understanding of how FastAPI allows them to serve APIs efficiently while keeping their application logic in Python. This approach ensures that analytical workflows can be shared across the organisation with minimal friction, while providing clear, maintainable, and production-ready code.

The sample repository can be found at: https://github.com/dl-lim/fastapi-example



```
FASTAPI-EXAMPLE [WSL: ...
> .venv
v app
  v api
    __init__.py
  v core
    security.py
    settings.py
  v db
    session.py
  v deps
    deps.py
  v models
    __init__.py
    item.py
    user.py
  v routers
    auth.py
    files.py
    items.py
    users.py
    ws.py
  v tests
    test_health.py
  __init__.py
  main.py
.gitignore
.python-version
app.db
main.py
pyproject.toml
README.md
uv.lock
```

app > main.py

```python
from contextlib import asynccontextmanager

from fastapi import BackgroundTasks, FastAPI
from fastapi.middleware.cors import CORSMiddleware

from app.api import api_router
from app.core.settings import settings
from app.db.session import init_db


@asynccontextmanager
async def lifespan(_: FastAPI):
    init_db()
    yield


app = FastAPI(
    title=settings.app_name,
    version="0.1.0",
    docs_url="/docs",
    redoc_url="/redoc",
    openapi_url="/openapi.json",
    lifespan=lifespan,
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=[origin.strip() for origin in settings.cors_origins.split(",")],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


@app.get("/health", tags=["health"], summary="Health check")
def health() -> dict:
    return {"status": "ok"}


@app.post("/tasks/send-email", tags=["background"], summary="Simulate sending email")
def send_email(background_tasks: BackgroundTasks, to: str) -> dict:
    def _send() -> None:
        # simulate side-effect
        return None

    background_tasks.add_task(_send)
    return {"queued": True, "to": to}


app.include_router(api_router)
```

App Startup

This section creates the FastAPI application, which is the core object that runs your API. It configures basic settings such as:

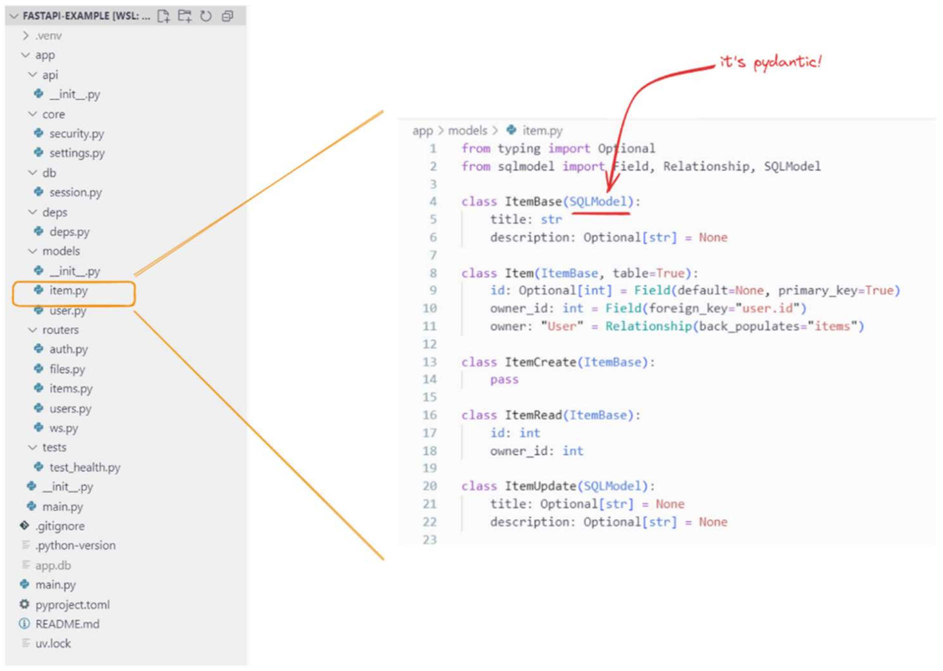
- The **name** and **version** of the API
- Where interactive documentation can be accessed (/docs for Swagger UI, /redoc for ReDoc)
- How the application handles **startup and shutdown events** via the lifespan parameter

This step initialises the application, preparing it to handle incoming requests and serve the functionality built by the data team. FastAPI handles much of the setup automatically, letting the team focus on building the endpoints and logic.

Middleware

Middleware is a layer that runs between the client and your API endpoints, processing requests and responses. In this example, the FastAPI application is adding **CORS (Cross-Origin Resource Sharing) middleware**.

This is a way to add reusable logic that runs before or after every request in your application. It can handle tasks such as logging, authentication, error handling, or modifying requests and responses.



```
FASTAPI-EXAMPLE [WSL: ...
> .venv
v app
  v api
    __init__.py
  v core
    security.py
    settings.py
  v db
    session.py
  v deps
    deps.py
  v models
    __init__.py
    item.py
    user.py
  v routers
    auth.py
    files.py
    items.py
    users.py
    ws.py
  v tests
    test_health.py
  __init__.py
  main.py
.gitignore
.python-version
app.db
main.py
pyproject.toml
README.md
uv.lock
```

app > models > item.py

```python
from typing import Optional
from sqlmodel import Field, Relationship, SQLModel

class ItemBase(SQLModel):
    title: str
    description: Optional[str] = None

class Item(ItemBase, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    owner_id: int = Field(foreign_key="user.id")
    owner: "User" = Relationship(back_populates="items")

class ItemCreate(ItemBase):
    pass

class ItemRead(ItemBase):
    id: int
    owner_id: int

class ItemUpdate(SQLModel):
    title: Optional[str] = None
    description: Optional[str] = None

```

SQLModel

SQLModel is a Python library that combines the best of two worlds: **Pydantic** for data validation and type enforcement, and **SQLAlchemy** for database ORM capabilities. This means you can define your data models once and use them both for interacting with your database and validating API input and output.

SQLModel is also developed in mind to support FastAPI natively.

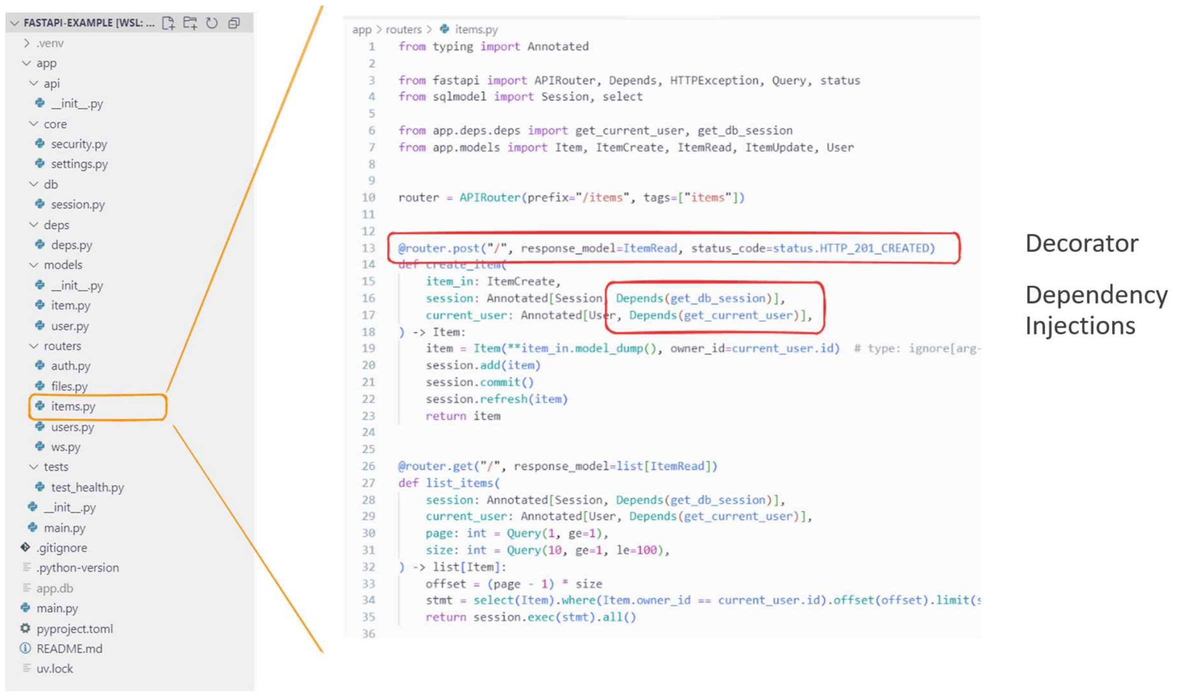
Why Pydantic matters:

- Ensures that all data entering or leaving your API conforms to expected types and formats (Enforces validation natively)
- Reduces runtime errors by catching inconsistencies early
- Makes your code self-documenting and easier to maintain

Why SQLModel is a great ORM:

- Simplifies database operations with Pythonic syntax
- Supports relationships, constraints, and primary/foreign keys naturally
- Works seamlessly with **Alembic** for database version control and migrations, allowing teams to track schema changes safely

For data teams, SQLModel makes it straightforward to manage database interactions while keeping data validation and application logic in Python, reducing context-switching and boilerplate code.



```
FASTAPI-EXAMPLE [WSL: ...
 > .venv
 v app
   v api
     __init__.py
   v core
     security.py
     settings.py
   v db
     session.py
   v deps
     deps.py
   v models
     __init__.py
     item.py
     user.py
   v routers
     auth.py
     files.py
     items.py
     users.py
     ws.py
   v tests
     test_health.py
   __init__.py
   main.py
 .gitignore
 .python-version
 app.db
 main.py
 pyproject.toml
 README.md
 uv.lock
```

app > routers > items.py

```python
from typing import Annotated

from fastapi import APIRouter, Depends, HTTPException, Query, status
from sqlmodel import Session, select

from app.deps.deps import get_current_user, get_db_session
from app.models import Item, ItemCreate, ItemRead, ItemUpdate, User


router = APIRouter(prefix="/items", tags=["items"])


@router.post("/", response_model=ItemRead, status_code=status.HTTP_201_CREATED)
def create_item(
    item_in: ItemCreate,
    session: Annotated[Session, Depends(get_db_session)],
    current_user: Annotated[User, Depends(get_current_user)],
) -> Item:
    item = Item(**item_in.model_dump(), owner_id=current_user.id)  # type: ignore[arg-
    session.add(item)
    session.commit()
    session.refresh(item)
    return item


@router.get("/", response_model=list[ItemRead])
def list_items(
    session: Annotated[Session, Depends(get_db_session)],
    current_user: Annotated[User, Depends(get_current_user)],
    page: int = Query(1, ge=1),
    size: int = Query(10, ge=1, le=100),
) -> list[Item]:
    offset = (page - 1) * size
    stmt = select(Item).where(Item.owner_id == current_user.id).offset(offset).limit(
    return session.exec(stmt).all()

```

Decorator

In FastAPI, creating a new route is as simple as adding a decorator to a function. The decorator defines the HTTP method, the endpoint path, and metadata such as the response model and status code.

This single line registers a function as a POST endpoint, automatically handling request parsing, response validation, and OpenAPI documentation generation in Swagger or Redoc.

Dependency Injection

Dependency injection in FastAPI is a way to provide reusable components or services to your route functions without manually creating them each time. You declare dependencies in the function signature, and FastAPI automatically provides the required objects when the endpoint is called.

- `session` is automatically provided by `get_db_session`, giving the function access to a database session
- `current_user` is provided by `get_current_user`, ensuring the function knows who is making the request

This pattern keeps your code clean, reusable, and testable, while abstracting away boilerplate setup logic for things like database connections or authentication.

7 CONCLUSION

Data teams operate at the intersection of specialised expertise and organisational demand. They must focus on developing accurate, reliable models and analyses while also ensuring that these outputs can be delivered to stakeholders efficiently. Meeting these external demands does not mean compromising on technical excellence or diverting attention to areas outside their core strengths.

Web development in Python provides a strategic solution to this challenge. By leveraging frameworks like FastAPI, data teams can expose APIs and services without leaving their primary language or workflow. This approach preserves the integrity of complex application logic, reduces inefficiencies from cross-language translation, and eliminates the need for unnecessary duplication of effort. At the same time, front-end development and user experience can remain the domain of specialised web teams, maintaining focus and quality across the stack.

If your team is in one of the following categories:

- Data
- AI/ML
- Actuarial
- Quantitative Modelling
- Finance
- Risk Management
- Cyber Security
- etc

... and are looking to expose your complex logic in Python to the web, a data mesh, or a web-based application, contact us and let us know what you're working on!

Get In Touch >

https://lunoxtech.com/get-in-touch

8 REFERENCES

Statistics Graphs:

https://www.statista.com/chart/16567/popular-programming-languages/

https://www.devoriales.com/post/374/python-tops-the-tiobe-index-the-most-popular-programming-languages-january-2025

https://coding-bootcamp-2021.acmbpdc.org/02-overview-of-programming-languages/

Repositories:

https://github.com/dl-lim/fastapi-example

https://github.com/django/django

https://github.com/pallets/flask

https://github.com/fastapi/fastapi

https://github.com/fastapi/sqlmodel