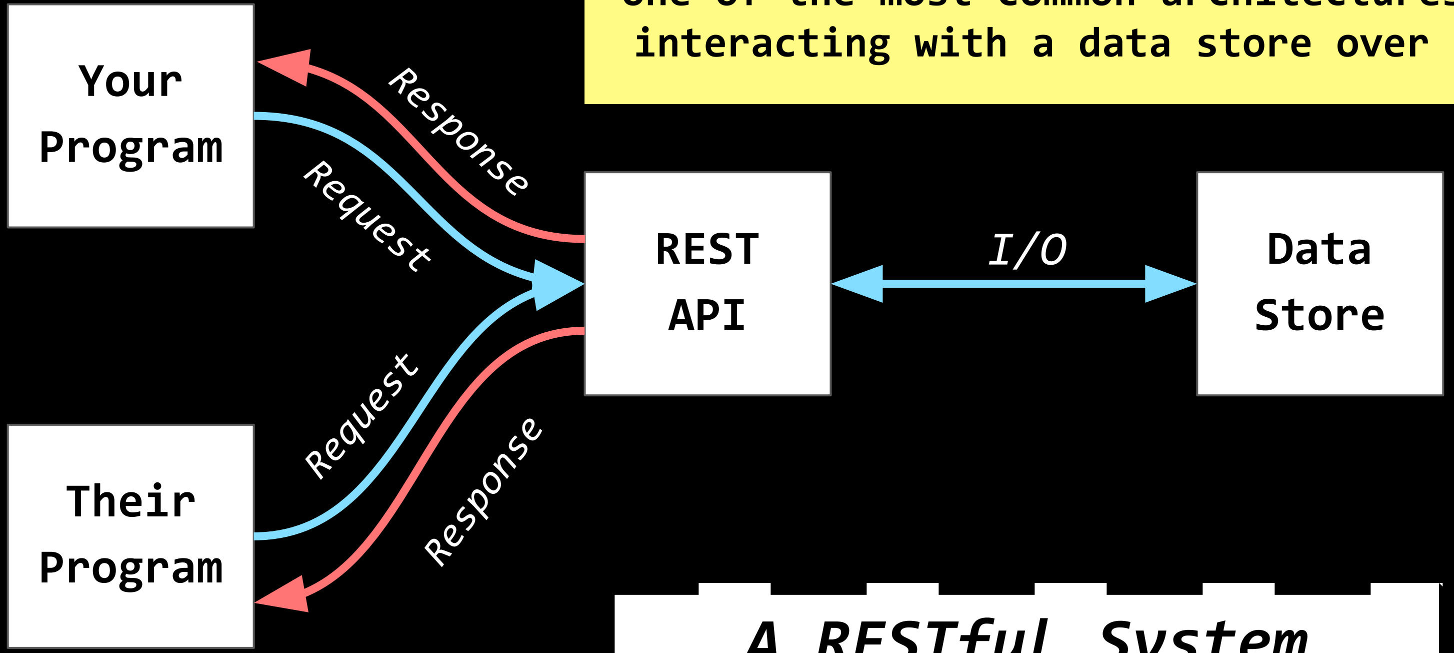
Anthony Holten
Sr. Software Enginer @ Interos

SQLModel

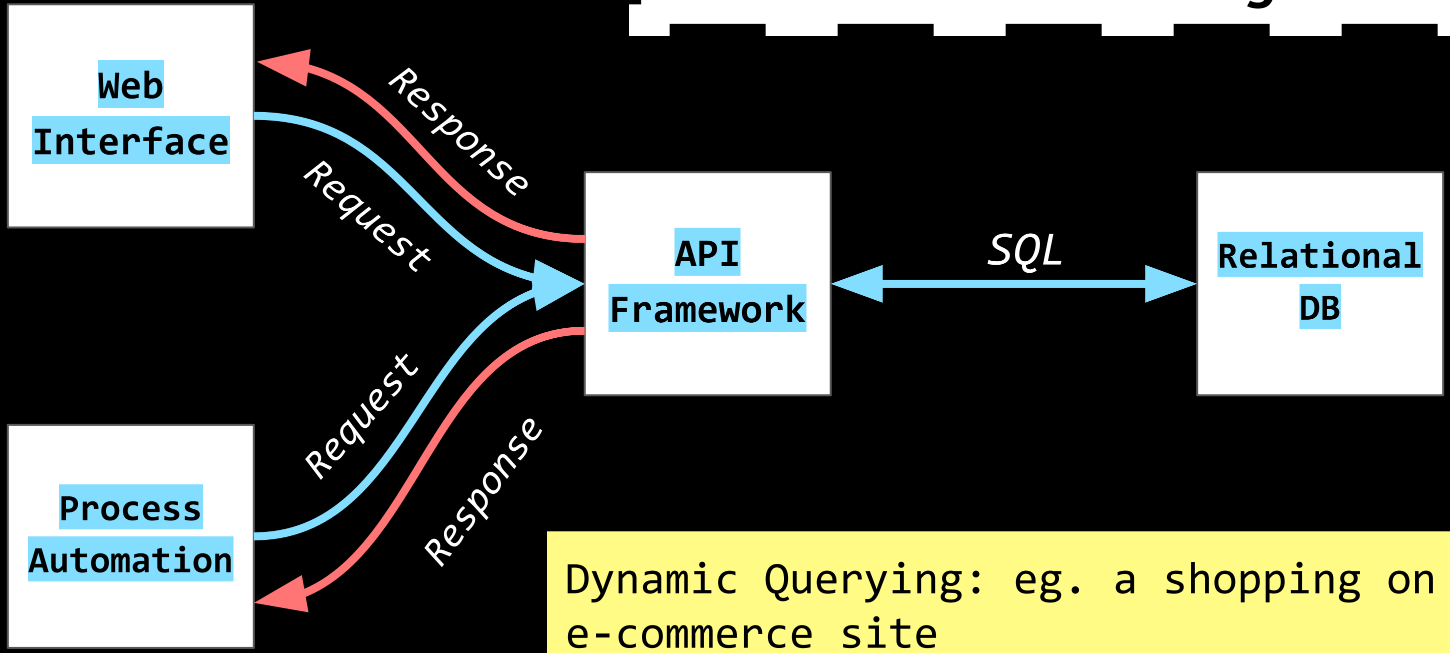
Easier state-of-the-art REST API Data Modeling in Python

REpresentational State Transfer (REST) is one of the most common architectures for interacting with a data store over HTTP



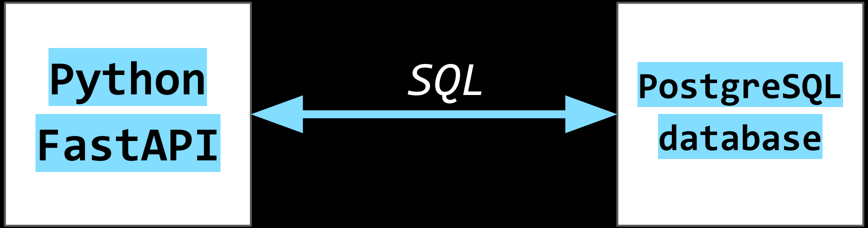
A RESTful System

A Common Design



Dynamic Querying: eg. a shopping on an e-commerce site

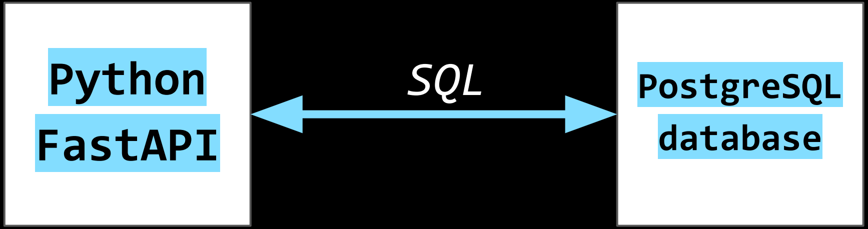
Programmatic Integration: eg. a flight comparison site, Internet of Things



You could template SQL…

Or you could use an Object Relation Mapper…

Either way, you’ll want to format and validate data



You could template SQL…

(native, Jinja, Mako)

Or you could use an Object Relation Mapper…

(Django, SQLAlchemy, Marshmallow)

(natively, Pydantic)

Either way, you’ll want to format and validate data

How do I validate Python objects?

How do I turn a relation into a python object?

A Pydantic class

A SQLAlchemy class

BIG Changes in 2023

Pydantic v2

June 30, 2023

Over a 20 minute read!

How long is the docs page about the update?

SQLAlchemy v2

January 26, 2023

Over a 1-hour read!

SQLModel

Easier state-of-the-art REST API Data Modeling in Python

SQLModel unifies both

Support added
v0.0.14
December 4, 2023

Write one
SQLModel class

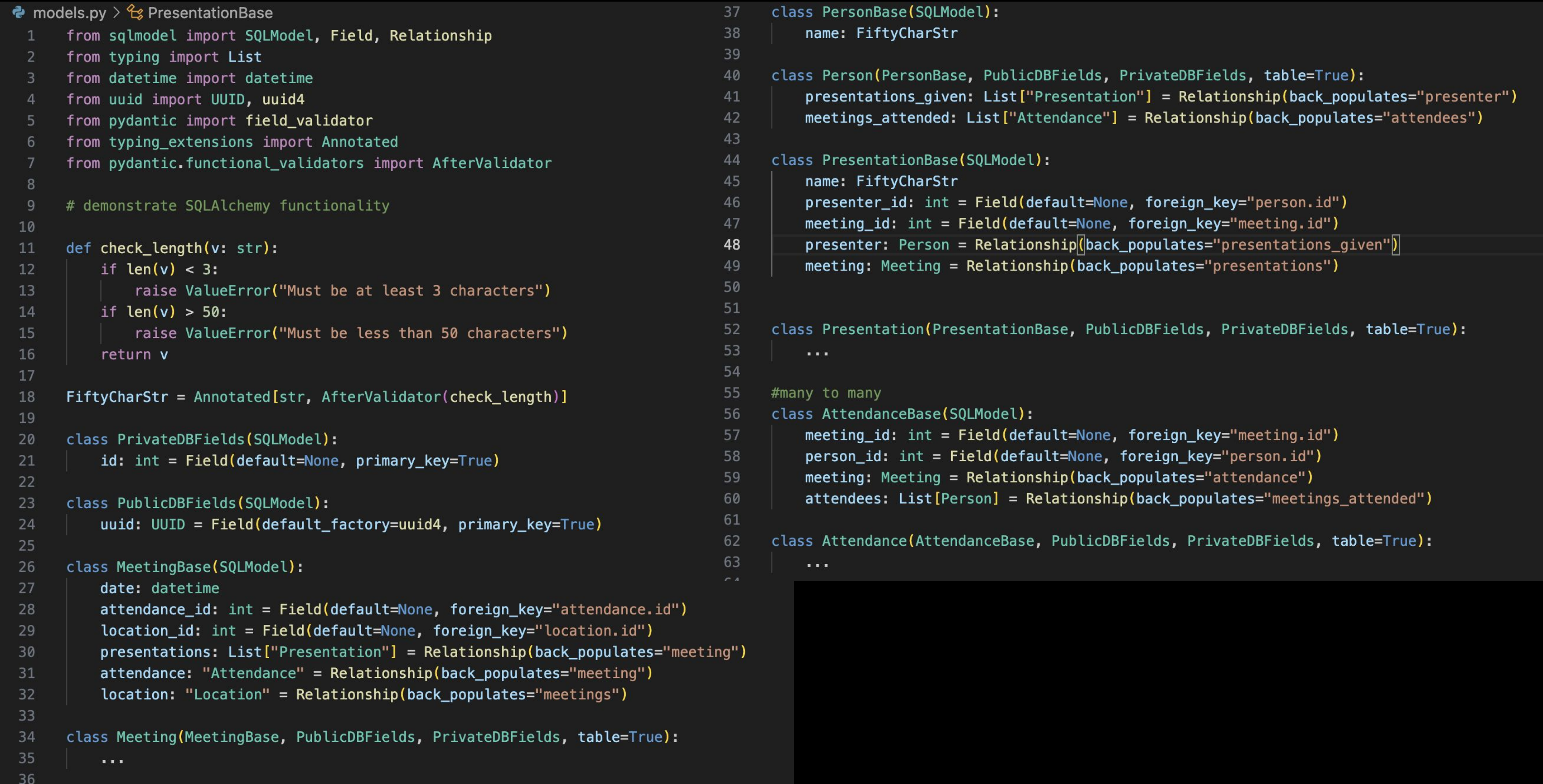
Support added
v0.0.14
November 18, 2023

Pydantic
v2

SQLAlchemy
v2

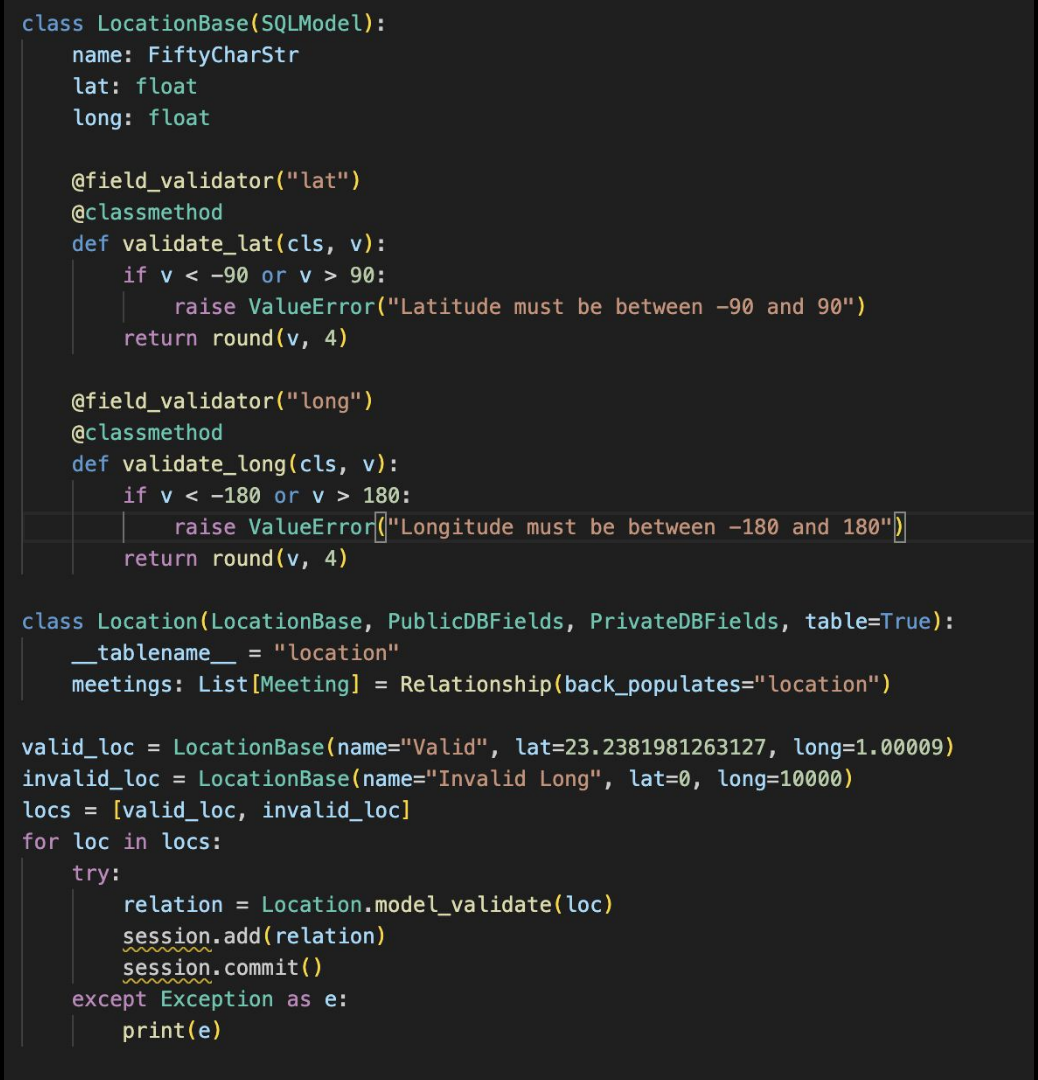
SQLModel

- Some of the latest ORM innovations
- Some of the latest data model validation innovations
- Turning your data class mapping into a table is as easy as setting kwarg table=true*
  - ** Except for when you have lots of special instructions to SQLAlchemy*
- Can map your data once instead of twice
  - *DRYer code -> less maintenance*
  - *DRYer code -> faster iteration*
- FastAPI compatibility is top of mind



```python
from sqlmodel import SQLModel, Field, Relationship
from typing import List
from datetime import datetime
from uuid import UUID, uuid4
from pydantic import field_validator
from typing_extensions import Annotated
from pydantic.functional_validators import AfterValidator

# demonstrate SQLAlchemy functionality

def check_length(v: str):
    if len(v) < 3:
        raise ValueError("Must be at least 3 characters")
    if len(v) > 50:
        raise ValueError("Must be less than 50 characters")
    return v

FiftyCharStr = Annotated[str, AfterValidator(check_length)]

class PrivateDBFields(SQLModel):
    id: int = Field(default=None, primary_key=True)

class PublicDBFields(SQLModel):
    uuid: UUID = Field(default_factory=uuid4, primary_key=True)

class MeetingBase(SQLModel):
    date: datetime
    attendance_id: int = Field(default=None, foreign_key="attendance.id")
    location_id: int = Field(default=None, foreign_key="location.id")
    presentations: List["Presentation"] = Relationship(back_populates="meeting")
    attendance: "Attendance" = Relationship(back_populates="meeting")
    location: "Location" = Relationship(back_populates="meetings")

class Meeting(MeetingBase, PublicDBFields, PrivateDBFields, table=True):
    ...
```

```python
class PersonBase(SQLModel):
    name: FiftyCharStr

class Person(PersonBase, PublicDBFields, PrivateDBFields, table=True):
    presentations_given: List["Presentation"] = Relationship(back_populates="presenter")
    meetings_attended: List["Attendance"] = Relationship(back_populates="attendees")

class PresentationBase(SQLModel):
    name: FiftyCharStr
    presenter_id: int = Field(default=None, foreign_key="person.id")
    meeting_id: int = Field(default=None, foreign_key="meeting.id")
    presenter: Person = Relationship(back_populates="presentations_given")
    meeting: Meeting = Relationship(back_populates="presentations")


class Presentation(PresentationBase, PublicDBFields, PrivateDBFields, table=True):
    ...

#many to many
class AttendanceBase(SQLModel):
    meeting_id: int = Field(default=None, foreign_key="meeting.id")
    person_id: int = Field(default=None, foreign_key="person.id")
    meeting: Meeting = Relationship(back_populates="attendance")
    attendees: List[Person] = Relationship(back_populates="meetings_attended")

class Attendance(AttendanceBase, PublicDBFields, PrivateDBFields, table=True):
    ...
```



```python
class LocationBase(SQLModel):
    name: FiftyCharStr
    lat: float
    long: float

    @field_validator("lat")
    @classmethod
    def validate_lat(cls, v):
        if v < -90 or v > 90:
            raise ValueError("Latitude must be between -90 and 90")
        return round(v, 4)

    @field_validator("long")
    @classmethod
    def validate_long(cls, v):
        if v < -180 or v > 180:
            raise ValueError("Longitude must be between -180 and 180")
        return round(v, 4)

class Location(LocationBase, PublicDBFields, PrivateDBFields, table=True):
    __tablename__ = "location"
    meetings: List[Meeting] = Relationship(back_populates="location")

valid_loc = LocationBase(name="Valid", lat=23.2381981263127, long=1.00009)
invalid_loc = LocationBase(name="Invalid Long", lat=0, long=10000)
locs = [valid_loc, invalid_loc]
for loc in locs:
    try:
        relation = Location.model_validate(loc)
        session.add(relation)
        session.commit()
    except Exception as e:
        print(e)
```

How do I know if SQLModel is right for me?

If you like…

Less library-specific code concepts, less SQL, more pythonic, more abstract

If you like…

A streamlined mapping syntax

If you have..

Many arbitrary shared fields across models