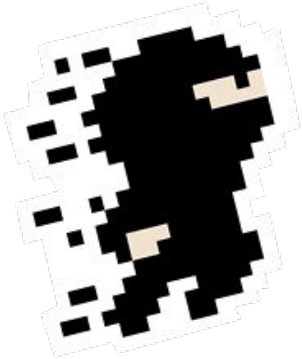


Django Ninja for API Development

Lessons from the Energy Market

PyCon 2024

Who are we?

Anastasiia Potekhina

- Backend developer at Helicon Technologies
- Self-taught developer
- Co-organizer of Pyladies Stockholm
- Dancing, roller skating, hiking

anapotekh

Beatriz Uezu

- Backend developer @ Helicon Technologies
- Specialized in Software Engineering
- Co-organizer of PyLadies Stockholm
- Photography, work out and running

beatrizuezu

Agenda

- What is Django Ninja and why did we chose it as our topic
- Challenges
- Motivations
- Type Hints
- Documentation
- Performance
- Code

What is Django Ninja?

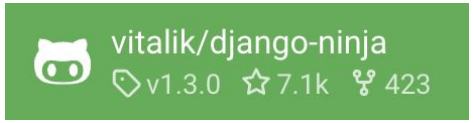
🌐 Modern API framework for Django that simplifies building fast and type-safe APIs

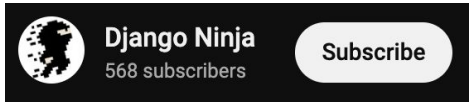
👤 Created by Vitaliy Kucheryaviy Django core contributor

📅 First released in 2020

🚀 Latest version 1.3.0

⚡ Heavily inspired by FastAPI (by Sebastián Ramírez)

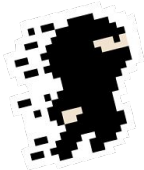




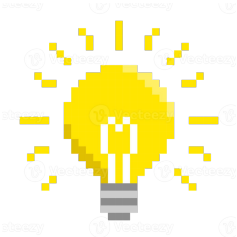
Goals:

- Easy to learn
- Fast in writing code
- Automatic docs
- Security
- High performance

Why Django Ninja?



- **Ninja-like stealth**: Highly effective but flying under the radar.



- **Perfect for work**: Fast, easy, and integrates seamlessly with Django.

- **Lightweight yet powerful**: Makes API development simple and enjoyable.

Challenges in API Development for Energy Sector

Complex Data Handling

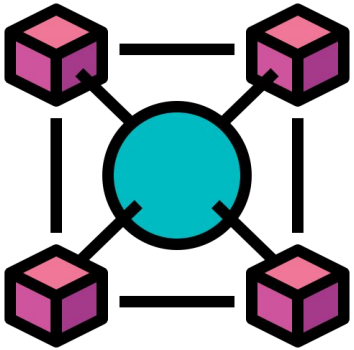
Hierarchical, interrelated data

Data Validation and Accuracy

Real-Time Data Streams

Challenges in API Development for Energy Sector

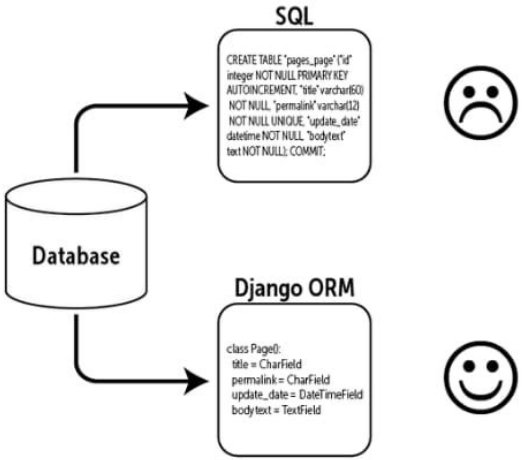
Performance at scale



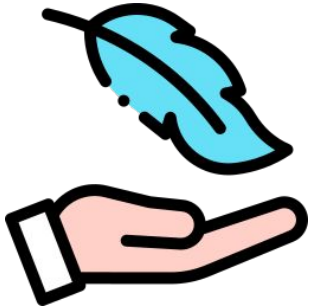
Interoperability with Legacy Systems

Security and Compliance

How Django Ninja helps to deal with those challenges



Pydantic



Pydantic

- Powered by type hints
- Fast
- JSON Schema
- Data conversion/strict mode if needed
- Functional validators and serializers
- Ecosystem

```python
from typing import List

def should_use(annotations: List[str]) -> bool:
    print("They're awesome!")
    return True
```

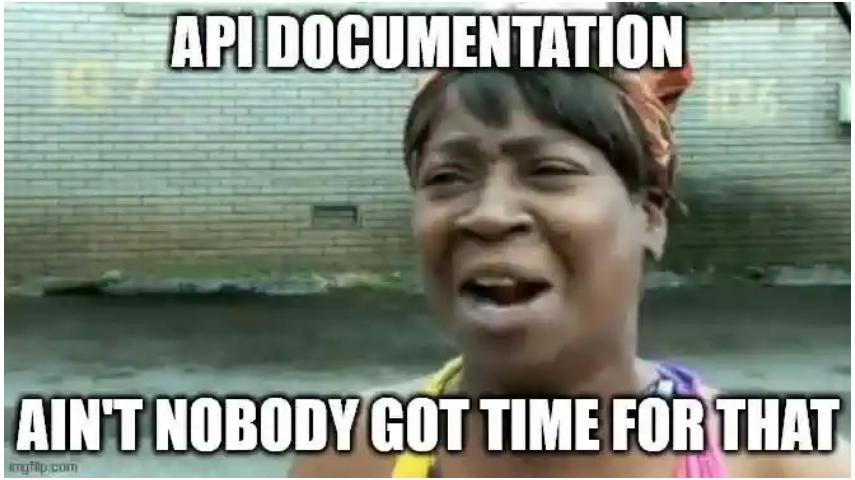
```python
from pydantic import BaseModel, validator

class User(BaseModel):
    name: str
    age: int

    @validator('age')
    def check_age(cls, value):
        if value < 0:
            raise ValueError('Age cannot be negative!')
        return value
```

How Django Ninja uses Pydantic

- Automatic Data Validation
- Type-Safety with Python Type Hints
- Simplified Serialization
- Error Handling

Swagger (OpenAPI standard) docs in Django Ninja

- Automatic API documentation
- Partially powered by Pydantic Models
- Interactive Interface
- Easy to use



Devices 1.0.0 OAS 3.1

/api/v1/openapi.json

default

POST /api/v1/measurement-devices Create Measurement Device

GET /api/v1/measurement-devices List Measurement Devices

GET /api/v1/measurements Retrieve Measurement Device

GET /api/v1/metrics Retrieve Metrics

Schemas

MeasurementDeviceResponse > Expand all object

MeasurementDeviceRequest > Expand all object

MeasurementResponse > Expand all object

MetricResponse > Expand all object

GET /api/v1/measurement-devices/{id} Retrieve Measurement Device

Parameters

Try it out

Name	Description
id * required string *(path)*	id

Responses

Code	Description	Links
200	OK Media type application/json Controls Accept header. Example Value | Schema	*No links*

```
{
  "id": 0,
  "name": "string",
  "location": "string",
  "created_at": "2024-10-29T10:28:54.888Z"
}
```

Schemas

MeasurementDeviceResponse ^ Collapse all object

- id > Expand all (integer | null)
- name* string <= 100 characters
- location > Expand all (string | null)
- created_at* string date-time

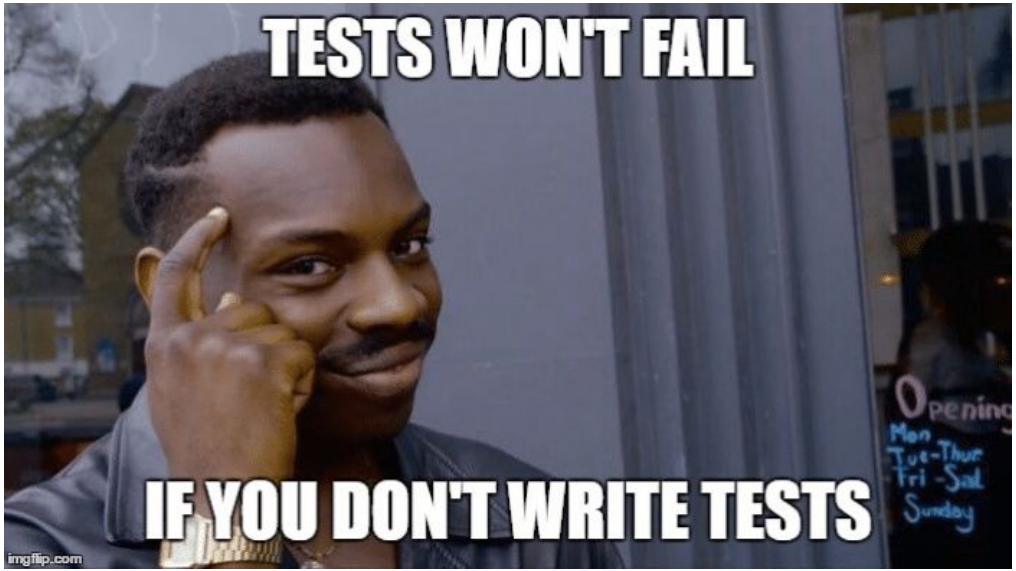
MeasurementDeviceRequest ^ Collapse all object

- name* string
- location* string

```python
class MeasurementDeviceRequest(Schema):
    name: str
    location: str

class MeasurementDeviceResponse(ModelSchema):
    class Meta:
        model = MeasurementDevice
        fields = ['id', 'name', 'location', 'created_at']
```

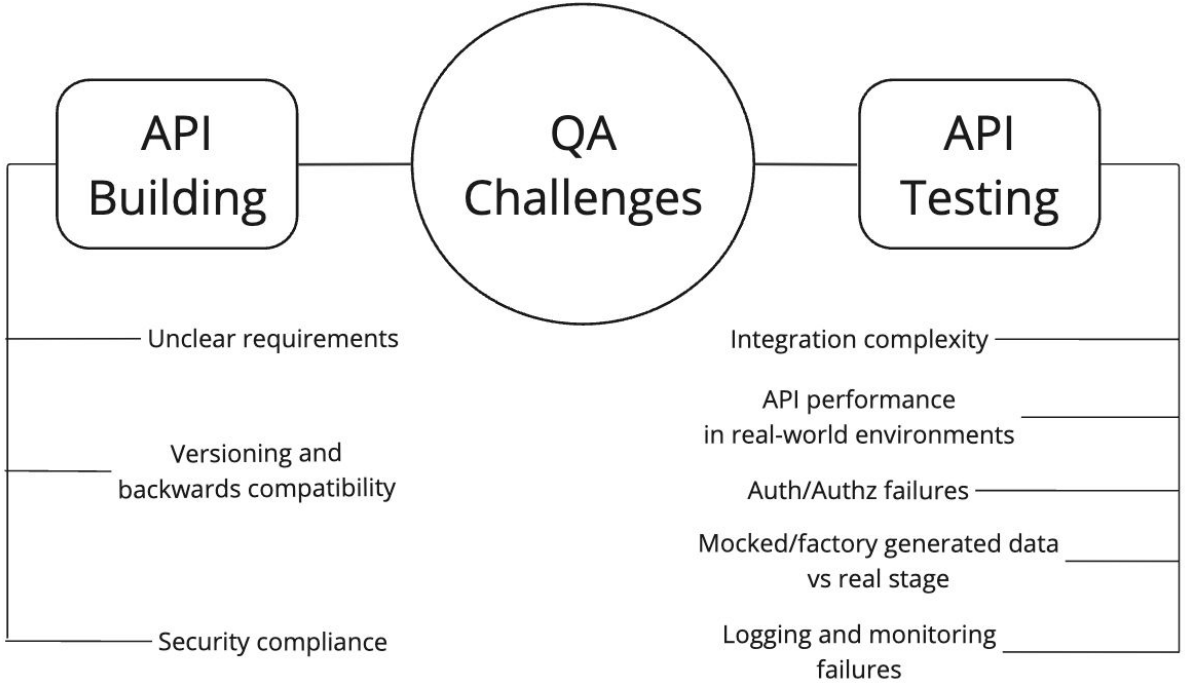
Devices 1.0.0 OAS 3.1

/api/v1/openapi.json



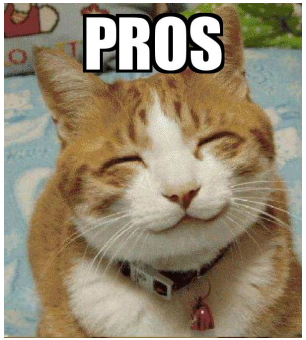
…it's always important how we handle testing during building process and afterwards. We should make sure to write comprehensive unit tests at early stage and then cover newly built API with integration tests in accordance with each request and business needs…

QA Insights on API Development for an Energy company

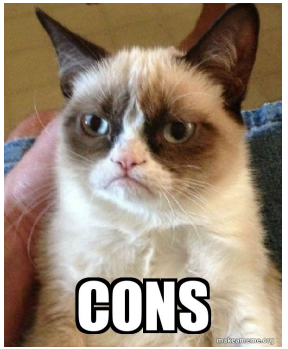


How to make it better?

Trade Offs

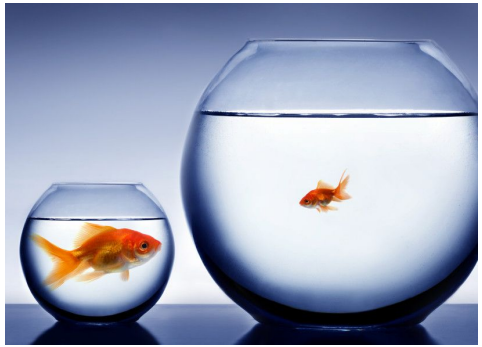


- Integration with Pydantic for type-safe models and validation, writing unit tests becomes more straightforward
- Automatic versioning support
- Django's built-in security protocols
- Integrates easily with real-time monitoring solutions like Prometheus
- lightweight and asynchronous capabilities make it well-suited for performance testing



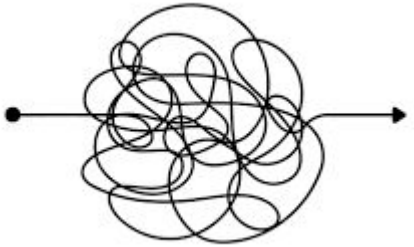
- Less mature ecosystem
- Async Testing might be complex
- Django dependency
- Pydantic integration might add unnecessary complexity

Key limitations of Django Ninja



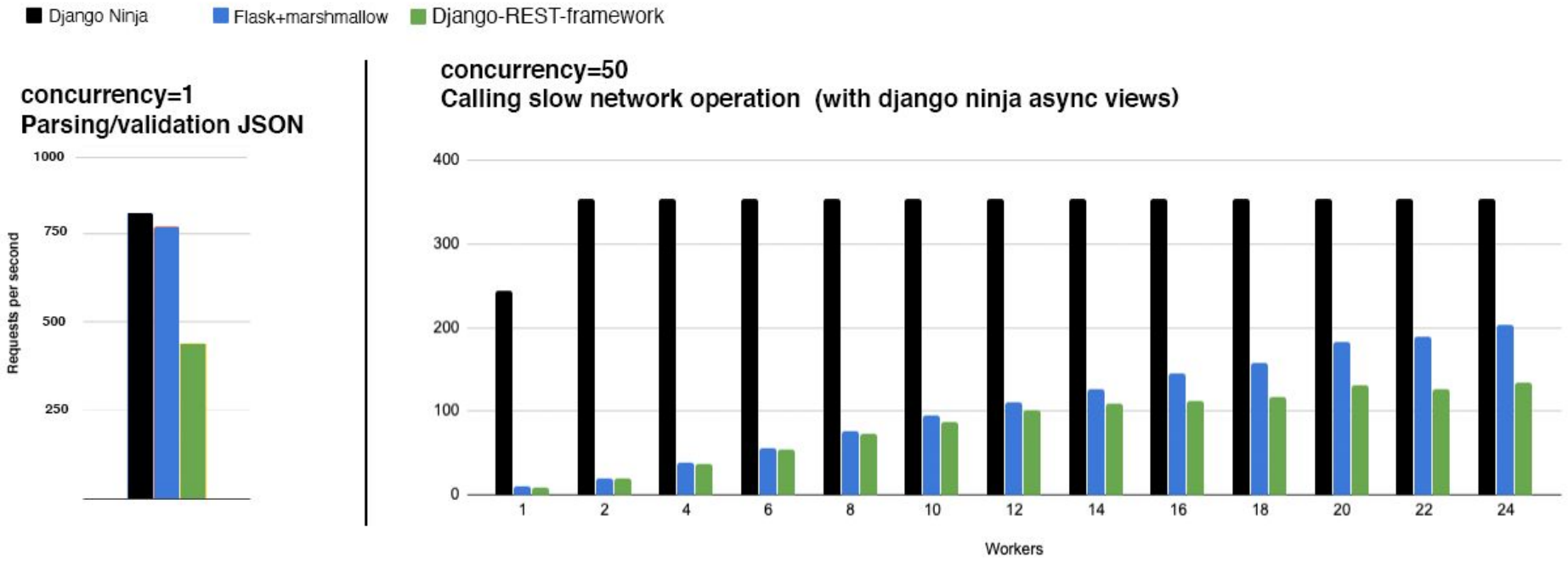
Small Ecosystem, Fewer built-in features

Limited Documentation



Async complexity

Performance



https://github.com/vitalik/django-ninja-benchmarks

Show me the code

beatrizuezu / **energy-service**

Service structure

api/views.py

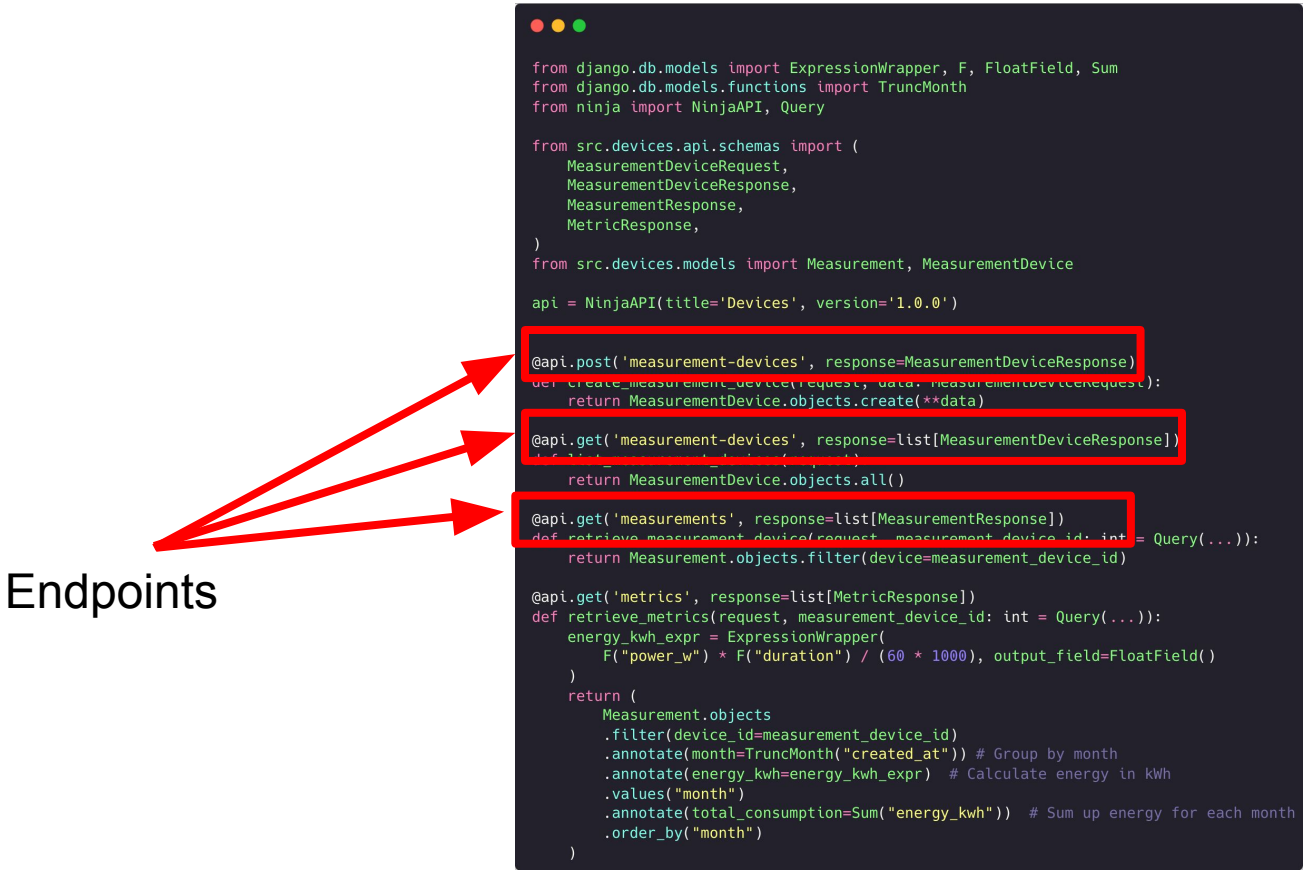
```python
from django.db.models import ExpressionWrapper, F, FloatField, Sum
from django.db.models.functions import TruncMonth
from ninja import NinjaAPI, Query

from src.devices.api.schemas import (
    MeasurementDeviceRequest,
    MeasurementDeviceResponse,
    MeasurementResponse,
    MetricResponse,
)
from src.devices.models import Measurement, MeasurementDevice

api = NinjaAPI(title='Devices', version='1.0.0')


@api.post('measurement-devices', response=MeasurementDeviceResponse)
def create_measurement_device(request, data: MeasurementDeviceRequest):
    return MeasurementDevice.objects.create(**data)

@api.get('measurement-devices', response=list[MeasurementDeviceResponse])
def list_measurement_devices(request):
    return MeasurementDevice.objects.all()

@api.get('measurements', response=list[MeasurementResponse])
def retrieve_measurement_device(request, measurement_device_id: int = Query(...)):
    return Measurement.objects.filter(device=measurement_device_id)

@api.get('metrics', response=list[MetricResponse])
def retrieve_metrics(request, measurement_device_id: int = Query(...)):
    energy_kwh_expr = ExpressionWrapper(
        F("power_w") * F("duration") / (60 * 1000), output_field=FloatField()
    )
    return (
        Measurement.objects
        .filter(device_id=measurement_device_id)
        .annotate(month=TruncMonth("created_at")) # Group by month
        .annotate(energy_kwh=energy_kwh_expr)  # Calculate energy in kWh
        .values("month")
        .annotate(total_consumption=Sum("energy_kwh"))  # Sum up energy for each month
        .order_by("month")
    )
```

api/views.py

Django Ninja app

```python
from django.db.models import ExpressionWrapper, F, FloatField, Sum
from django.db.models.functions import TruncMonth
from ninja import NinjaAPI, Query

from src.devices.api.schemas import (
    MeasurementDeviceRequest,
    MeasurementDeviceResponse,
    MeasurementResponse,
    MetricResponse,
)
from src.devices.models import Measurement, MeasurementDevice

api = NinjaAPI(title='Devices', version='1.0.0')


@api.post('measurement-devices', response=MeasurementDeviceResponse)
def create_measurement_device(request, data: MeasurementDeviceRequest):
    return MeasurementDevice.objects.create(**data)

@api.get('measurement-devices', response=list[MeasurementDeviceResponse])
def list_measurement_devices(request):
    return MeasurementDevice.objects.all()

@api.get('measurements', response=list[MeasurementResponse])
def retrieve_measurement_device(request, measurement_device_id: int = Query(...)):
    return Measurement.objects.filter(device=measurement_device_id)

@api.get('metrics', response=list[MetricResponse])
def retrieve_metrics(request, measurement_device_id: int = Query(...)):
    energy_kwh_expr = ExpressionWrapper(
        F("power_w") * F("duration") / (60 * 1000), output_field=FloatField()
    )
    return (
        Measurement.objects
        .filter(device_id=measurement_device_id)
        .annotate(month=TruncMonth("created_at")) # Group by month
        .annotate(energy_kwh=energy_kwh_expr)  # Calculate energy in kWh
        .values("month")
        .annotate(total_consumption=Sum("energy_kwh"))  # Sum up energy for each month
        .order_by("month")
    )
```

api/views.py


Endpoints
from django.db.models import ExpressionWrapper, F, FloatField, Sum
from django.db.models.functions import TruncMonth
from ninja import NinjaAPI, Query
from src.devices.api.schemas import (
MeasurementDeviceRequest,
MeasurementDeviceResponse,
MeasurementResponse,
MetricResponse,
)
from src.devices.models import Measurement, MeasurementDevice
api = NinjaAPI(title='Devices', version='1.0.0')
@api.post('measurement-devices', response=MeasurementDeviceResponse)
return MeasurementDevice.objects.create(**data)
@api.get('measurement-devices', response=list[MeasurementDeviceResponse])
return MeasurementDevice.objects.all()
@api.get('measurements', response=list[MeasurementResponse])
return Measurement.objects.filter(device=measurement_device_id)
@api.get('metrics', response=list[MetricResponse])
def retrieve_metrics(request, measurement_device_id: int = Query(...)):
energy_kwh_expr = ExpressionWrapper(
F("power_w") * F("duration") / (60 * 1000), output_field=FloatField()
)
return (
Measurement.objects
.filter(device_id=measurement_device_id)
.annotate(month=TruncMonth("created_at")) # Group by month
.annotate(energy_kwh=energy_kwh_expr) # Calculate energy in kWh
.values("month")
.annotate(total_consumption=Sum("energy_kwh")) # Sum up energy for each month
.order_by("month")
)

api/views.py

```python
from django.db.models import ExpressionWrapper, F, FloatField, Sum
from django.db.models.functions import TruncMonth
from ninja import NinjaAPI, Query

from src.devices.api.schemas import (
    MeasurementDeviceRequest,
    MeasurementDeviceResponse,
    MeasurementResponse,
    MetricResponse,
)
from src.devices.models import Measurement, MeasurementDevice

api = NinjaAPI(title='Devices', version='1.0.0')


@api.post('measurement-device', response=MeasurementDeviceResponse)
def create_measurement_device(request, data: MeasurementDeviceRequest):
    return MeasurementDevice.objects.create(**data)

@api.get('measurement-devices', response=list[MeasurementDeviceResponse])
def list_measurement_devices(request):
    return MeasurementDevice.objects.all()

@api.get('measurements', response=list[MeasurementResponse])
def retrieve_measurement_device(request, measurement_device_id: int = Query(...)):
    return Measurement.objects.filter(device=measurement_device_id)

@api.get('metrics', response=list[MetricResponse])
def retrieve_metrics(request, measurement_device_id: int = Query(...)):
    energy_kwh_expr = ExpressionWrapper(
        F("power_w") * F("duration") / (60 * 1000), output_field=FloatField()
    )
    return (
        Measurement.objects
        .filter(device_id=measurement_device_id)
        .annotate(month=TruncMonth("created_at")) # Group by month
        .annotate(energy_kwh=energy_kwh_expr)  # Calculate energy in kWh
        .values("month")
        .annotate(total_consumption=Sum("energy_kwh"))  # Sum up energy for each month
        .order_by("month")
    )
```

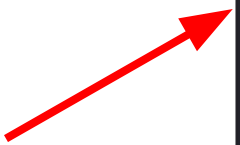
Request schema

api/views.py

Response schema

```python
from django.db.models import ExpressionWrapper, F, FloatField, Sum
from django.db.models.functions import TruncMonth
from ninja import NinjaAPI, Query

from src.devices.api.schemas import (
    MeasurementDeviceRequest,
    MeasurementDeviceResponse,
    MeasurementResponse,
    MetricResponse,
)
from src.devices.models import Measurement, MeasurementDevice

api = NinjaAPI(title='Devices', version='1.0.0')


@api.post('measurement-devices', response=MeasurementDeviceResponse)
def create_measurement_device(request, data: MeasurementDeviceRequest):
    return MeasurementDevice.objects.create(**data)

@api.get('measurement-devices', response=list[MeasurementDeviceResponse])
def list_measurement_devices(request):
    return MeasurementDevice.objects.all()

@api.get('measurements', response=list[MeasurementResponse])
def retrieve_measurement_device(request, measurement_device_id: int = Query(...)):
    return Measurement.objects.filter(device=measurement_device_id)

@api.get('metrics', response=list[MetricResponse])
def retrieve_metrics(request, measurement_device_id: int = Query(...)):
    energy_kwh_expr = ExpressionWrapper(
        F("power_w") * F("duration") / (60 * 1000), output_field=FloatField()
    )
    return (
        Measurement.objects
        .filter(device_id=measurement_device_id)
        .annotate(month=TruncMonth("created_at")) # Group by month
        .annotate(energy_kwh=energy_kwh_expr)  # Calculate energy in kWh
        .values("month")
        .annotate(total_consumption=Sum("energy_kwh"))  # Sum up energy for each month
        .order_by("month")
    )
```

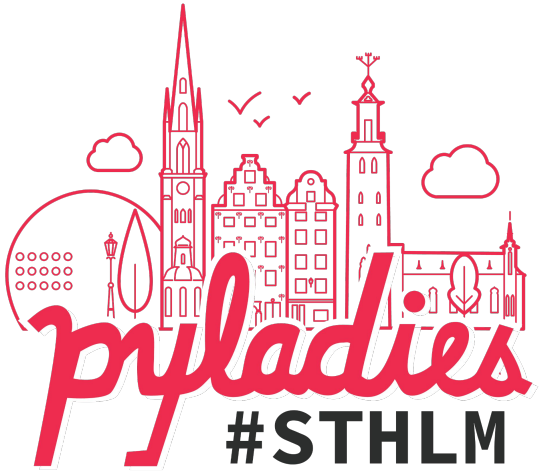
api/schemas.py



It uses Pydantic
under the hood

```python
from datetime import date

from ninja import ModelSchema, Schema

from src.devices.models import Measurement, MeasurementDevice


class MeasurementDeviceRequest(Schema):
    name: str
    location: str

class MeasurementDeviceResponse(ModelSchema):
    class Meta:
        model = MeasurementDevice
        fields = ['id', 'name', 'location', 'created_at']


class MeasurementResponse(ModelSchema):
    class Meta:
        model = Measurement
        fields = ['device', 'power_w', 'duration', 'created_at']


class MetricResponse(Schema):
    month: date
    total_consumption: float
```

Join us at PyLadies Stockholm!



Next event:

Meetup together with Stockholm Python Users Group
November 25, 17:30
at SUNET

THANK YOU

anapotekh

beatrizuezu