Web Data

REST, Serialization, Views

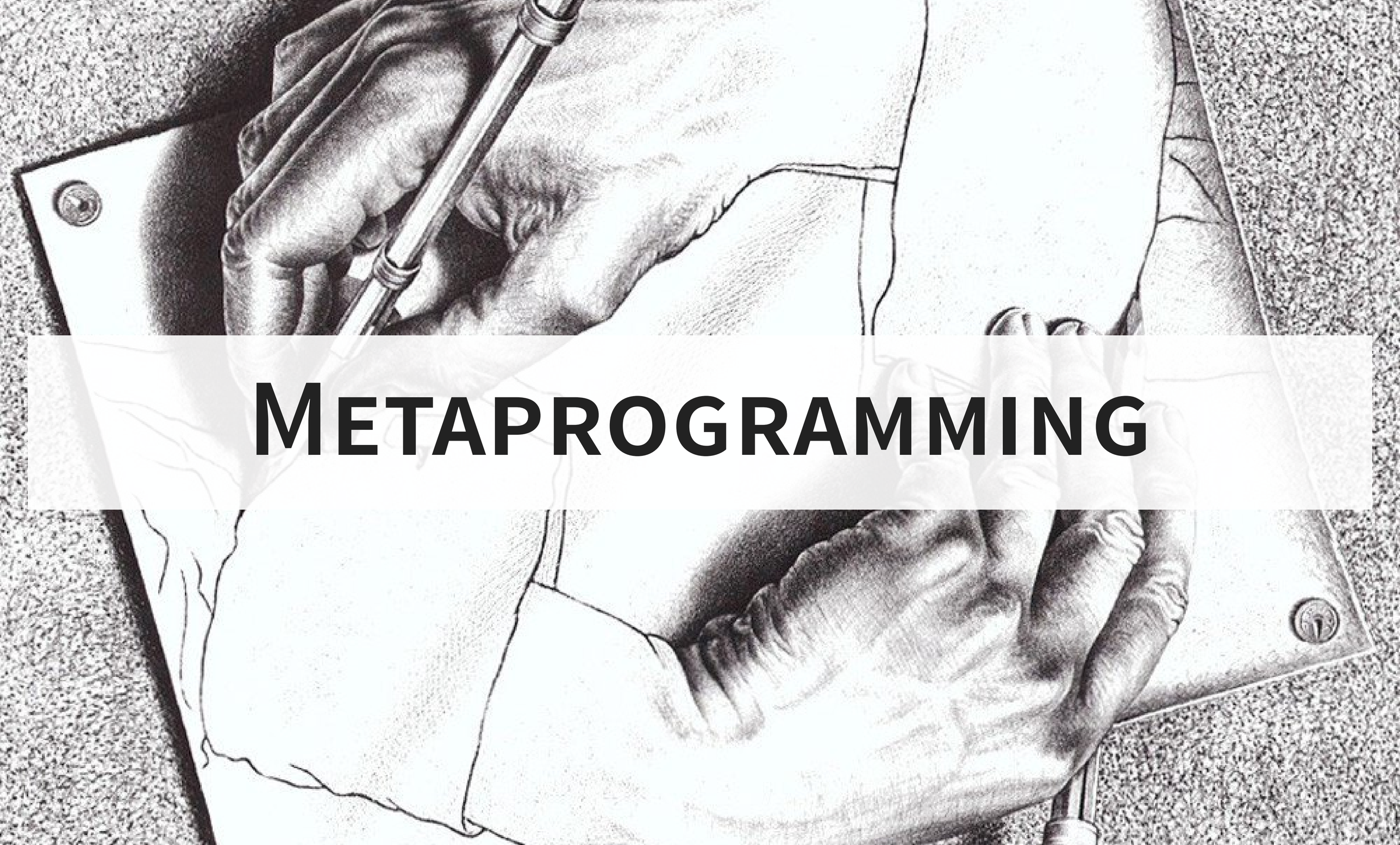
Dr. Scott Gorlin

Harvard University

Fall 2018

Agenda

- Metaprogramming
- Atomic Targets
- Ser/Deser
- DB Design
- DB Juggling
- The Web
- API’s
- Reading



Metaprogramming

Metaprogramming

> *Metaprogramming is a programming technique in which computer programs have the ability to treat programs as their data. It means that a program can be designed to read, generate, analyse or transform other programs, and even modify itself while running*
>
> — *Wikipedia*

We've already seen this...

```python
@unpack_argparse
def some_func(name='scott', **kwargs):
    # Old: some_func(args):
    # print(args.name)
    ...
```

```python
def unpack_argparse(func):
    @wraps
    def wrapped(args):
        return func(**{
            k: getattr(args, k)
            for k in dir(args)
            if not k.startswith('_')
        })
    return wrapped
```

```python
if __name__ == '__main__':
    parser = ArgumentParser()
    parser.add_args(
        '-n', '--name',
        default='scotty')
    some_func(parser.parse_args())
```

A decorator is ***so meta***.

We've already seen this...

```
build([MyTask()])
dask_df.groupby().sum().compute()
```

When graph libraries modify the graph at runtime, they are meta programming

We've already seen this...

```
# environment.yml
name: {{cookiecutter.project_slug}}
dependencies:
  - jupyter
{%- if cookiecutter.use_pytest == 'y' %}
  - pytest
{%- endif %}
  - python={{cookiecutter.python_version}}
  - setuptools_scm
```

Templated programs via jinja, cookiecutter, etc

Metaclasses

Everything is an object, even classes!

```
class A:
    def x(self):
        return 5
>>> a = A(); a.x()
```

… is the same as

```
A = type('A', tuple(), {'x': lambda self: 5})

>>> a = A(); a.x()
5
```

a is an instance of A, and A is an instance of type!

Metaclasses

We can subclass `type` to alter class creation!

```
class MyMeta(type):
    def __new__(cls, name, bases, dct):
        """Creates a new class!"""
        c = super().__new__(
            cls, name, bases, dct)
        c.x = lambda self: 5
        return c
```

```
class A(metaclass=MyMeta):
    pass

>>> a = A(); a.x()
5
```

This looks like it could be done with a decorator… except it runs ***for all new subclasses too***

Metaclasses

```
class Timeable(type):

    def __new__(cls, name, bases, dct):
        dct.update({
            'time_'+k: timeit(v)
            for k, v in dct.items()
            if callable(v)
        })
        return super().__new__(
            cls, name, bases, dct)
```

```
class MLModel(metaclass=Timeable):
    def train(self, x, y):
        ...

>>> MLModel().time_train(x, y)
```

Metaclasses

```
class Property:
    pass
```

```
class Encapsulated(type):
    def __new__(cls, name, bases, dct):

        props = [
            k for k, v in dct.items()
            if isinstance(v, Property)
        ]
        for p in props:
            dct['get_' + p] = (
                lambda self: getattr(
                    self, '_' + p, None
                )
            )
        ...
```

```
class Point(metaclass=Encapsulated):
    x = Property()

>>> p = Point(); p.set_x(5)
>>> p.get_x()
5
```

Metaclasses

You likely don't need metaclasses (until you do)

But they are critical to understand complex code

Model Inheritance

```
class CommonInfo(Model):
    name = CharField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    ...
```

… one `Student` table, with all the columns

```
class Place(Model):
    name = CharField()

class Restaurant(Place):
    ...
```

… two tables; `Restaurant` FK's to `Place` because a restaurant is a kind of place

```
class Person(Model):
    name = CharField()

    def get_name(self):
        return self.name
```

```
class PigLatinPerson(Person):
    class Meta:
        proxy = True

    def get_name(self):
        n = self.name
        return ''.join([
            n[1].upper(), n[2:],
            n[0].lower(), 'ay'
        ])
```

… just the `Person` table

Model Options

```
class CommonInfo(Model):
    name = CharField()

    class Meta:
        abstract = True
```

```
class Student(CommonInfo):
    # Inherits `name`
    grade = FloatField()
```

How does django know `Student` is a real table? It should have inherited `abstract = True`!

ModelBase

```
class ModelBase(type):
    def __new__(cls, name, bases, attrs):
        ...
        # Lots of magic for declarative options
        attr_meta = attrs.pop('Meta', None)
        ...

class Model(metaclass=ModelBase):
    ...
```

ModelBase extensively tweaks each new Model to ensure the fields work properly and to alter the standard inheritance patterns to be more intuitive for DB design

The Point

Metaprogramming allows you to write a Domain Specific Language, big or small

You can isolate the logic (the metaprogram) from the data (the actual program)

Atomic Targets

Atomicity

Transactional

```
with transaction.atomic():
    # NB: this is only for the default DB!
    MyModel.objects.create(...)
```

SQL gives us clean all-or-nothing transactions (usually)

Workflow

```
class DjangoModelTarget(Target):
    def exists(self):
        # Check if row exists??
```

Luigi wants a target to complete atomically

Database Targets

```
# luigi.contrib.postgres
class PostgresTarget(Target):
    def __init__(self, update_id, **conn_keywords):
        ...

    def exists(self):
        # Check *metadata*: did query run?
        "SELECT 1 from {marker_table}"
        " where update_id={self.update_id}"
```

Luigi has builtin `sqlalchemy` `Target`'s and `Query` tasks for Postgres, Redshift, and generic RDBMS

The `update_id` can be used to implement a salted workflow!

Relying on an 'existence' table violates a pure 'can I read this target' form of atomicity

Can queries modify rows, or just append, or must they support both? What about unique keys?

Database Targets

```
class DjangoModelTarget(Target):
    def __init__(self, model, **unique):
        ...
```

```
    def exists(self):
      try:
        # Assumes there is a unique
        # constraint
        m = self.model.objects.get(
            **self.unique_kwargs
        )
        return True
    except self.model.DoesNotExist:
        return False
```

You can easily write a target to check for existence of a `unique` (or `unique_together`) key

Database Targets

```
class SaltedDjangoModelTarget(Target):
    def __init__(self, model, version, **unique):
        ...
```

```
    def exists(self):
      try:
          # Assumes there is a unique
          # constraint
          m = self.model.objects.get(
              version = self.version,
              **self.unique_kwargs
          )
          return True
      except self.model.DoesNotExist:
          return False
```

You can easily write a target to check for existence of a `unique` (or `unique_together`) key

You can even support salted targets

pk	name	version
1	Scott	8a3cef1
2	Scott	c091ef1

Multiple Rows

```
class MultiTarget(DjangoTarget):
    def exists(self):
        return self.model.filter().count() > 0

with atomic():
    target.model.bulk_create(...)
```

If any two targets have mutually exclusive rows, we can continue the pattern to multi-row, non-unique inserts

Except if a valid task should insert 0 rows, eg an empty dataset, and you want to know it is empty!

SER/DESER

Python Objects

Everything is a mutable dict!

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
p = Point(1, 2)
p.adsf = 5  # Works just fine
p.__dict__  # See inside
```

... unless it's a tuple or implements `__slots__`

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(1, 2)
p.x # 1
p.s = 14
AttributeError: 'Point' object has no attribute 's'
```

Python Objects

Everything is a mutable dict!

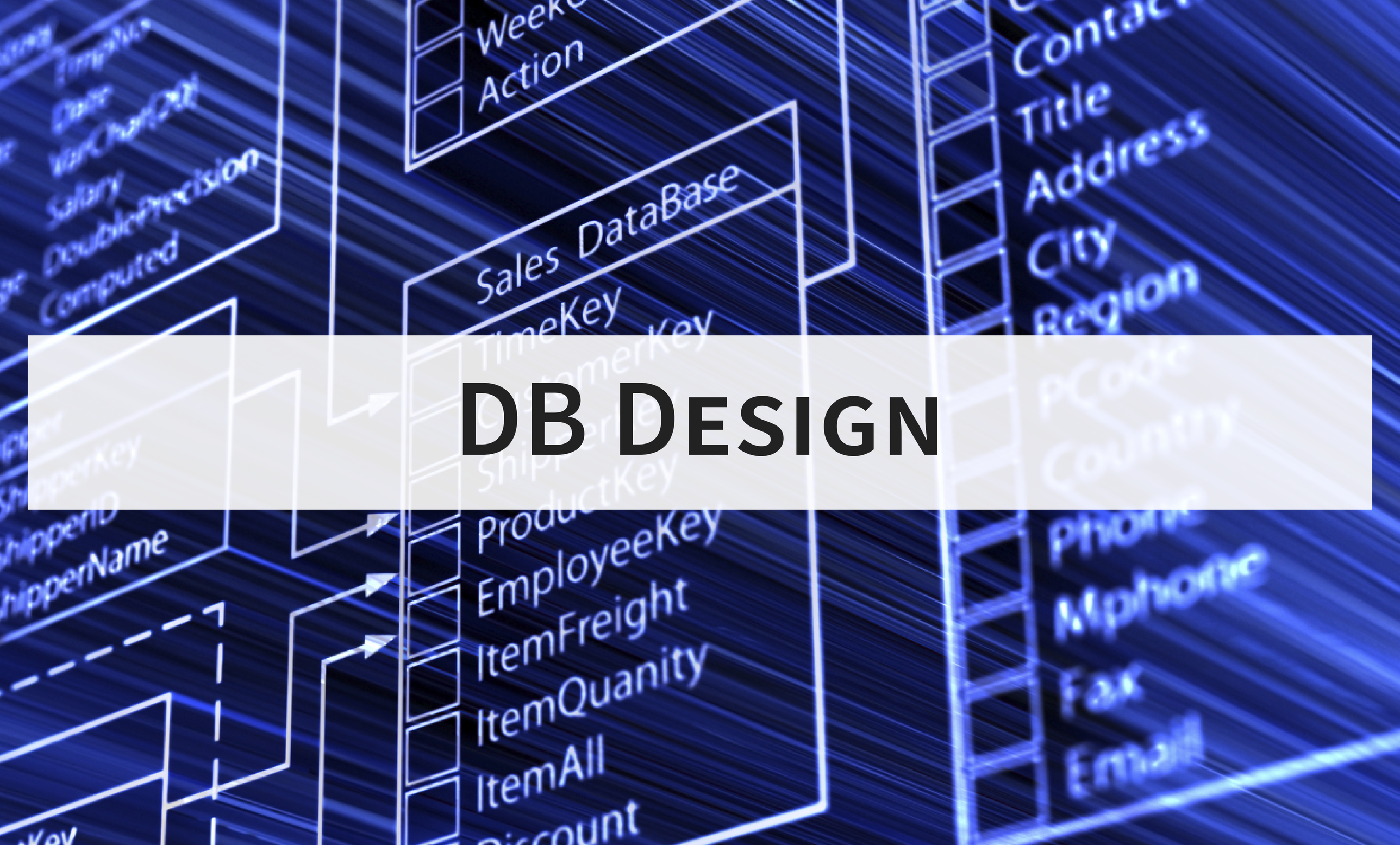
Mutable types make for easy coding, at some overhead cost

Sometimes it is worth working with tuples for performance

```
>>> values = ['id', 'name']
>>> Blog.objects.values(*values)
<QuerySet [{'id': 1, 'name': 'Beatles Blog'}, ...]>
```

```
>>> Blog.objects.values_list(*values)
<QuerySet [(1, 'Beatles Blog'), ...]>
>>> DataFrame(_, columns=values)
   id          name
0   1  Beatles Blog
```

Note how the schema isn't replicated in every record!


Sales DataBase
TimeKey
ProductKey
EmployeeKey
ItemFreight
ItemQuanity
ItemAll
Discount
Action
Salary
Computed
ShipperName
Title
Address
City
Region
Phone
Mphone
Fax
Email

DB Design

Normalization

Denormalized

student	assignment	grade
Scott	Pset 1	89
August	Pset 1	91

Normalized Forms reduce redundancy and may increase (or decrease) query complexity

Normalized

student_id	assignment_id	grade
1	1	89
2	1	91

id	name
1	Scott
2	August

id	assignment
1	Pset 1

Normalization

> *Normalization entails organizing the columns (attributes) and tables (relations) of a database to ensure that their dependencies are properly enforced by database integrity constraints.*
>
> — *Wikipedia*

Normalization

Denormalized

```
class Grade(Model):
    student = CharField()
    assignment = CharField()
    grade = FloatField()
```

```
Grade.objects.values(
    'student'
).annotate(grade=Avg('grade'))
```

Fast and easy, but how to rename a pset?

Normalized

```
class Assignment(Model):
    assignment = CharField()

class Student(Model):
    name = CharField()

class Grade(Model):
    student = ForeignKey(Student)
    assignment = ForeignKey(Assignment)
    grade = FloatField()
```

How to query?

Normalization

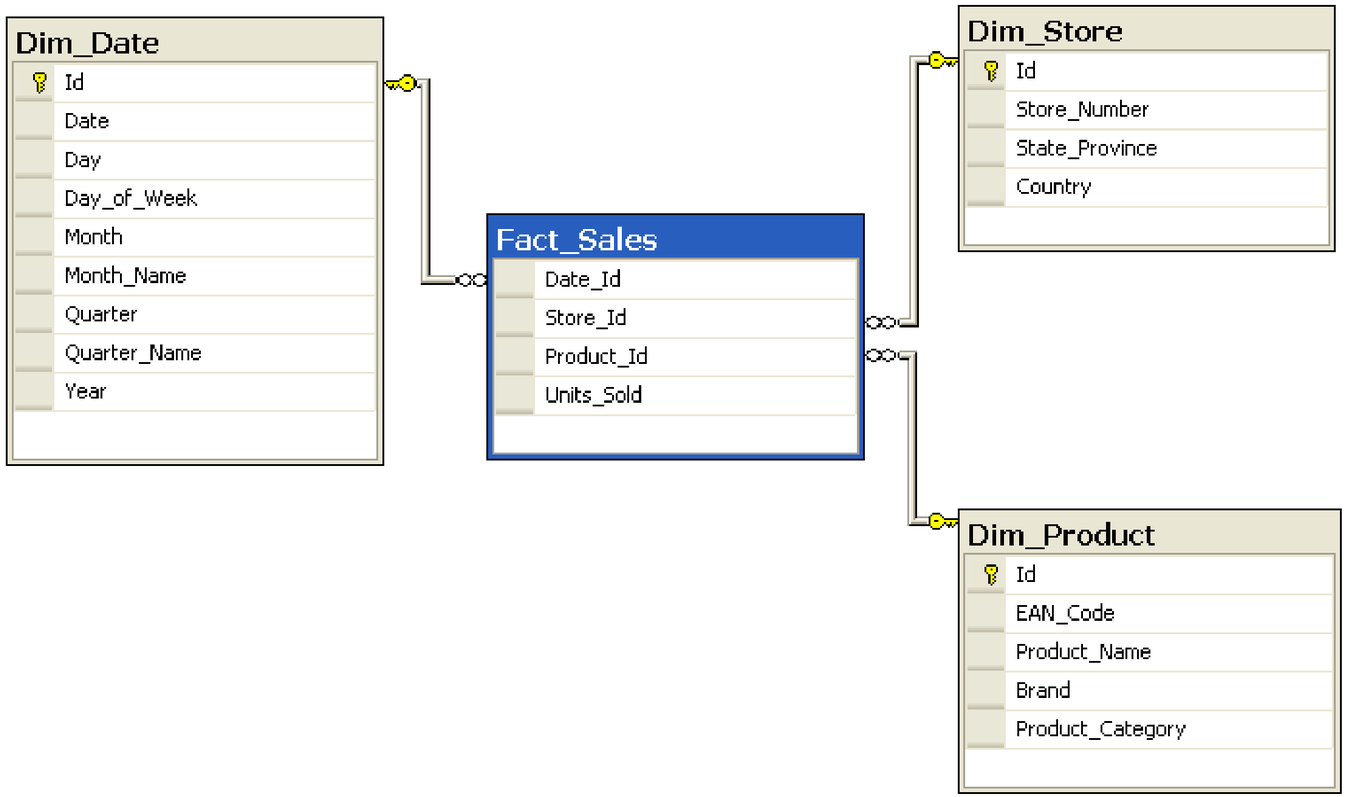
Normalized

```
class Assignment(Model):
    assignment = CharField()

class Student(Model):
    name = CharField()

class Grade(Model):
    student = ForeignKey(Student)
    assignment = ForeignKey(Assignment)
    grade = FloatField()
```

How to query?

```
def vl(*args):
    return Grade.objects.values_list(*args)

# What you'd expect
>>> vl('student').annotate(grade=Avg('grade'))
[(Student('scott'), 89), ...]

# Fast and minimal, avoid join
>>> vl('student_id').annotate(...)
[(1, 89), ...]

# Join on 'real' value
>>> vl('student__name').annotate(...)
[('scott', 89), ...]
```

Stars and Flakes



Stars and Flakes

Fact Tables

… are designed for aggregations

Every business ‘fact’ is captured in exactly one row (as an event, or as an aggregate)

Columns are summary stats (count, price, totals, etc)

Dimension Tables

… are designed for groups and filters

Fact tables will FK to Dim tables to qualify rows

Stars and Flakes

Fact Tables

```
class SalesFact(Model):
    # NB: Not DateField!
    date = ForeignKey(DateDim)
    product = ForeignKey(ProductDim)

    # Facts (theory should be just 1)
    units = IntField()
    revenue = FloatField()

total_sales = SalesFact.objects.sum()
```

Dim Tables

```
class DateDim(Model):
    date = DateField()
    year = IntField()
    season = CharField()
    quarter = IntField()
    is_holiday = BooleanField()
```

Dims may store derived properties!

Compare:

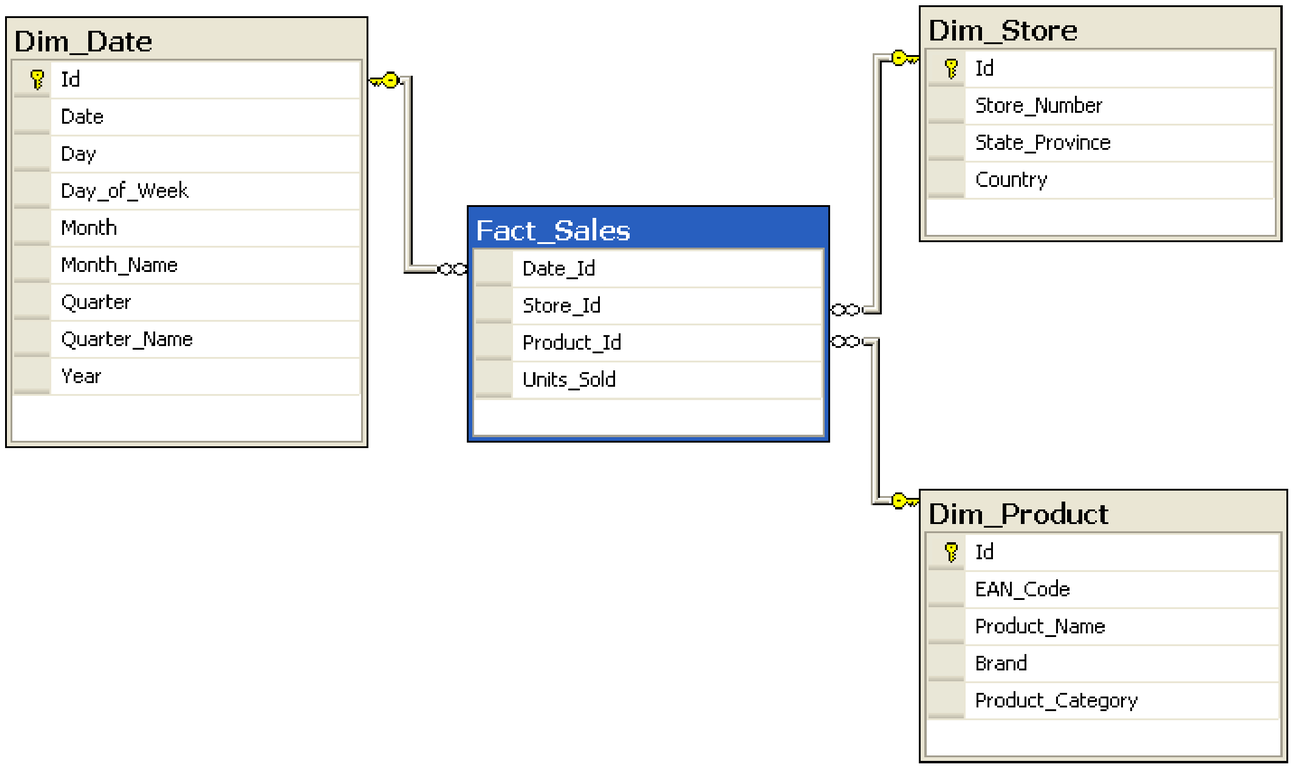
```
SalesFact.objects.filter(
    date__year=2018,        # Join and lookup, vs
    date__date__year=2018   # sql-side YEAR(date)
).sum()
```

Star Schema

Fact FK's to 1 level of Dim tables

Dim tables may have derivative properties to make easy querying

Common DataMart design due to fast writing

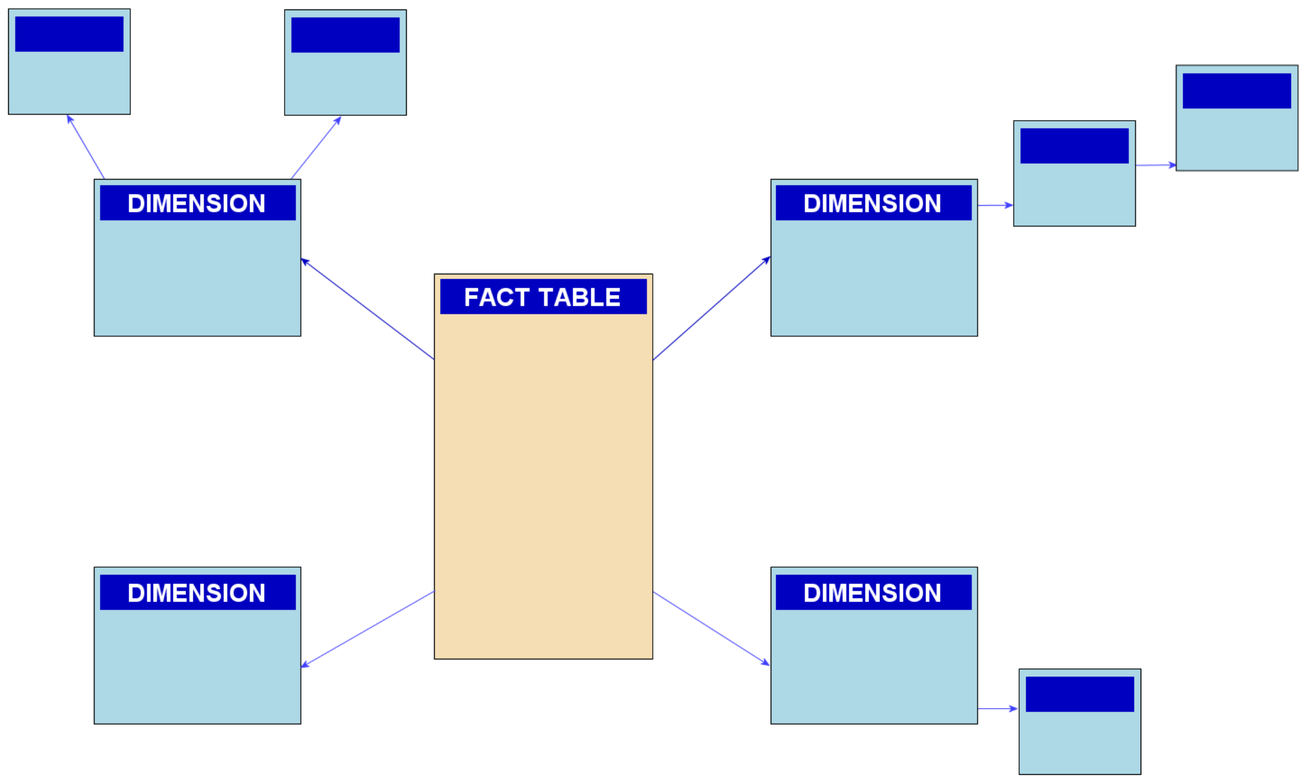


Star schemas are highly denormalized and usually don't guarantee unique dims! Avoid querying dim's by PK directly!

Snowflake Schema

Fully normalized dimensional variant of Star

More referential integrity, but harder to query



Intelligent Keys

The primary key on a table is usually auto-incrementing and best considered ‘private’

But, Eng teams (and you) can optimize it

Beware over-optimization! This can kill readability

```
class DateDim(Model):
    date = DateField()
    def save(**kwargs):
        if not self.pk:
            self.pk = self.get_pk(self.date)
        super(DateDim, self).save(**kwargs)

    @classmethod
    def get_pk(cls, date):
        return int(date.strftime('%Y%m%d'))
```

```
SalesFact.objects.filter(
    date_id=DateDim.get_pk(date.today())
).sum()
```

… or can be a critical performance boost, eg for ***partitions*** by date. SQL can’t optimize queries across the join

DB Juggling

Multiple DB's

External DB's

It's common to rely on other DB's you don't own…

eg provided by Eng.

Try to keep ***your*** tables in one DB!

Write an isolated app for each DB

```
class ExternalTable(Model):
    class Meta:
        # You may set this to True
        # during unit tests!
        managed = False
```

Multiple DB's

External DB's

Use ***per-app routers*** to configure!

Write an isolated app for each DB

```
class PerAppRouter(object):
    """Specifies a DB for an entire app

    In your ```settings.py```, add::

        ROUTING_TABLE = {
            'my_app':'my_db'
        }

    """
```

```
    def db_for_read(self, model, **hints):
        from django.conf import settings
        return settings.ROUTING_TABLE.get(
            model._meta.app_label, 'default')
        ...
```

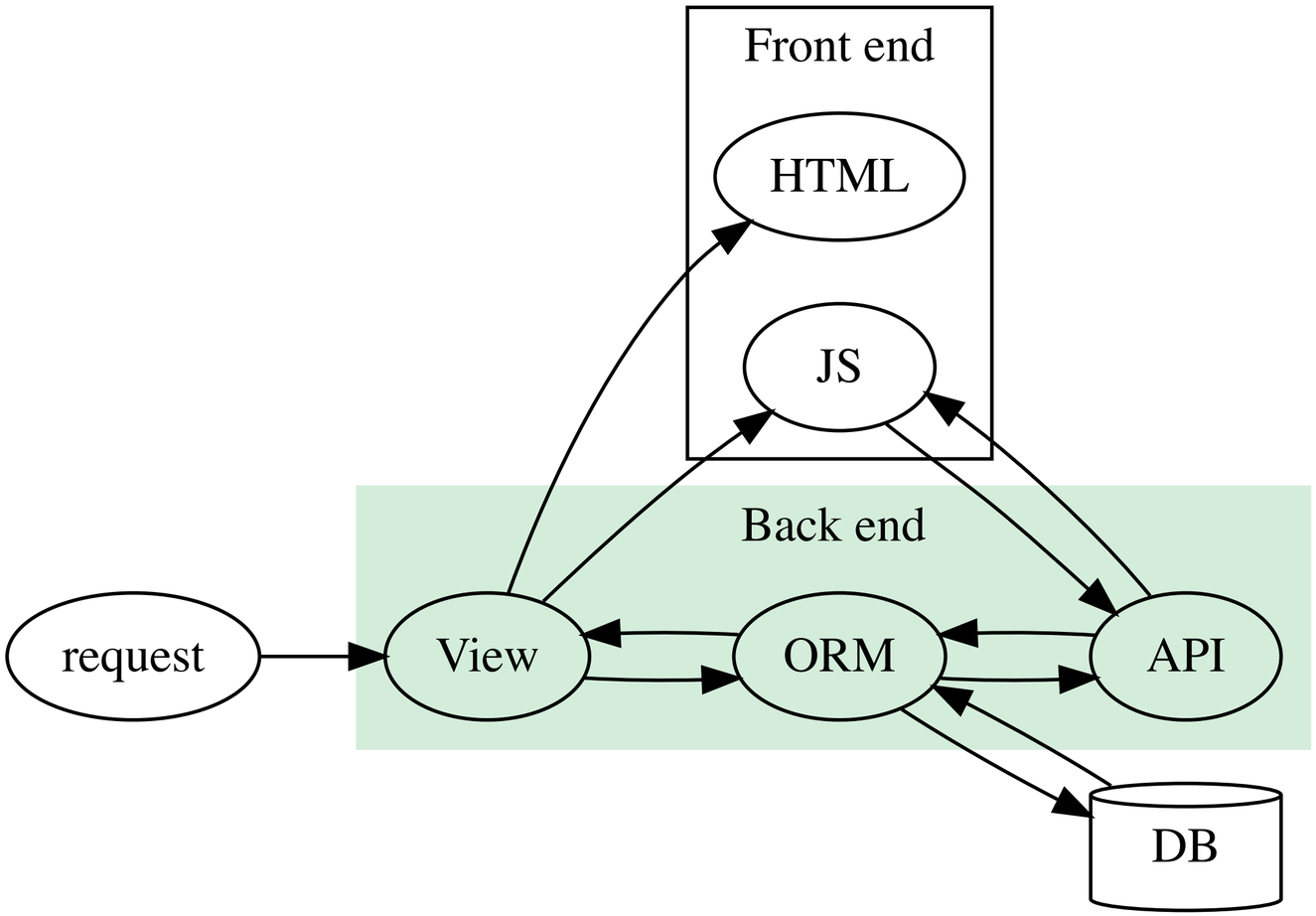
Atomics

Django assumes the default DB everywhere. If you're writing elsewhere, stick with sane defaults and patterns

```
def atomic_for_model(model, **kwargs):
    """Transactions for the db on a specific model

    Usage::

        with atomic_for_model(MyModel):
            ...

    """

    model_db = model.objects.all().db

    return atomic(using=model_db, **kwargs)
```

The Web

Page Lifecyle



Django Views

URL routing

```
from django.urls import path

from . import views

urlpatterns = [
    path('articles/<int:poll_id>/', views.detail),
]
```

The View

```
def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body>
</html>" % now
    return HttpResponse(html)
```

```
def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render(request, 'polls/detail.html',
{'poll': p})
```

Django Views

It’s easy to shoot out a quick web page w/ a template and some python data wrangling

Django provides a few other batteries like forms, some template tools, etc

Besides the built-in admin, Django does ***not*** provide a default frontend

Bootstrap

Build responsive projects on the web with the world's most popular front-end component library.

Bootstrap is an open source toolkit for developing with HTML, CSS, and JS.

Bootstrap

The Missing Frontend Library

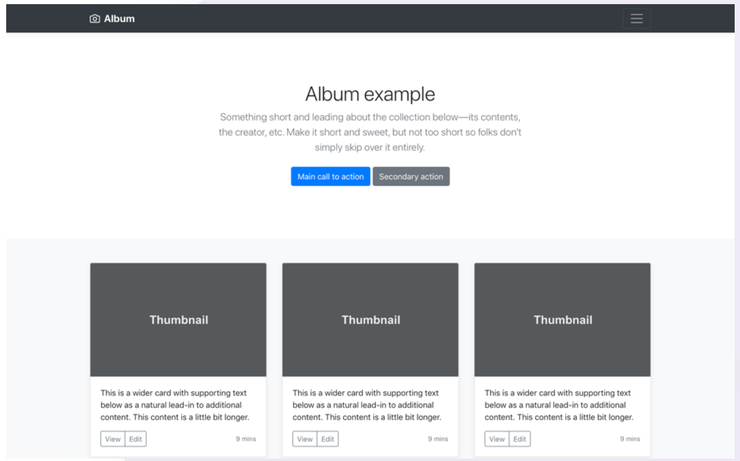
The missing ‘make it look awesome’ frontend stack

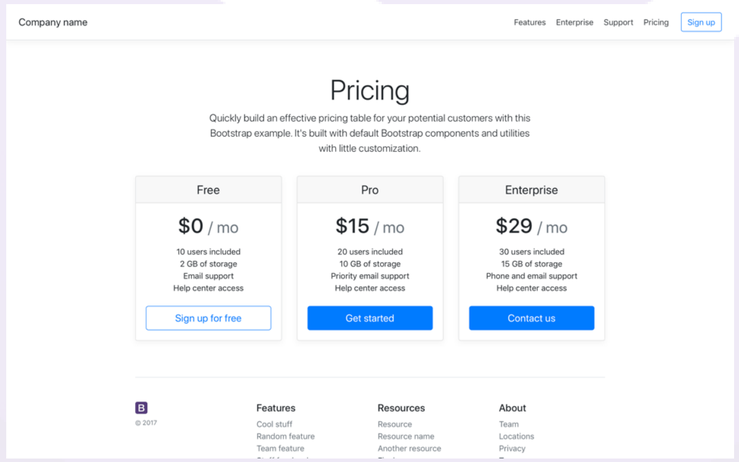
https://getbootstrap.com/

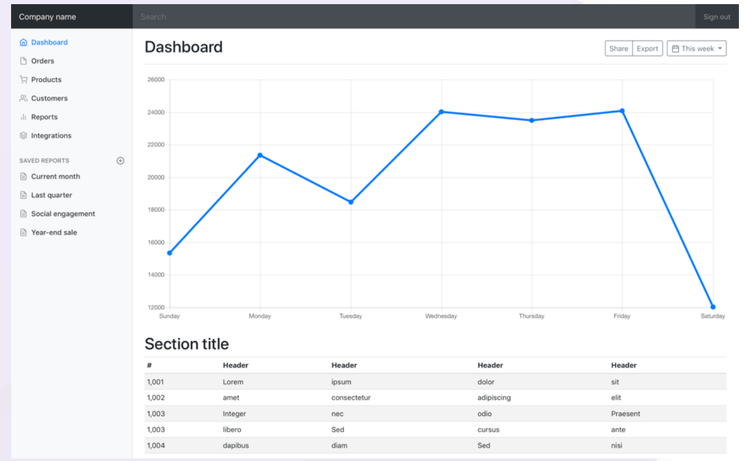
I stole these alert themes from them!

Plus popups, layout, etc

Bootstrap


Album
Album example
Something short and leading about the collection below—its contents, the creator, etc. Make it short and sweet, but not too short so folks don't simply skip over it entirely.
Main call to action
Secondary action
Thumbnail
This is a wider card with supporting text below as a natural lead-in to additional content. This content is a little bit longer.
View
Edit
9 mins
Thumbnail
This is a wider card with supporting text below as a natural lead-in to additional content. This content is a little bit longer.
View
Edit
9 mins
Thumbnail
This is a wider card with supporting text below as a natural lead-in to additional content. This content is a little bit longer.
View
Edit
9 mins


Company name
Features
Enterprise
Support
Pricing
Sign up
Pricing
Quickly build an effective pricing table for your potential customers with this Bootstrap example. It's built with default Bootstrap components and utilities with little customization.
Free
$0 / mo
10 users included
2 GB of storage
Email support
Help center access
Sign up for free
Pro
$15 / mo
20 users included
10 GB of storage
Priority email support
Help center access
Get started
Enterprise
$29 / mo
30 users included
15 GB of storage
Phone and email support
Help center access
Contact us
© 2017
Features
Cool stuff
Random feature
Team feature
Resources
Resource
Resource name
Another resource
About
Team
Locations
Privacy


Company name
Search
Sign out
Dashboard
Orders
Products
Customers
Reports
Integrations
SAVED REPORTS
Current month
Last quarter
Social engagement
Year-end sale
Dashboard
Share
Export
This week
26000
24000
22000
20000
18000
16000
14000
12000
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Section title
Header Header Header Header
1,001 Lorem ipsum dolor sit
1,002 amet consectetur adipiscing elit
1,003 Integer nec odio Praesent
1,003 libero Sed cursus ante
1,004 dapibus diam Sed nisi

… But!

Old School Cool

Server-generated webpages are so 2005

… even if written in python!

You don’t want Chrome to sit there spinning while the ORM works!

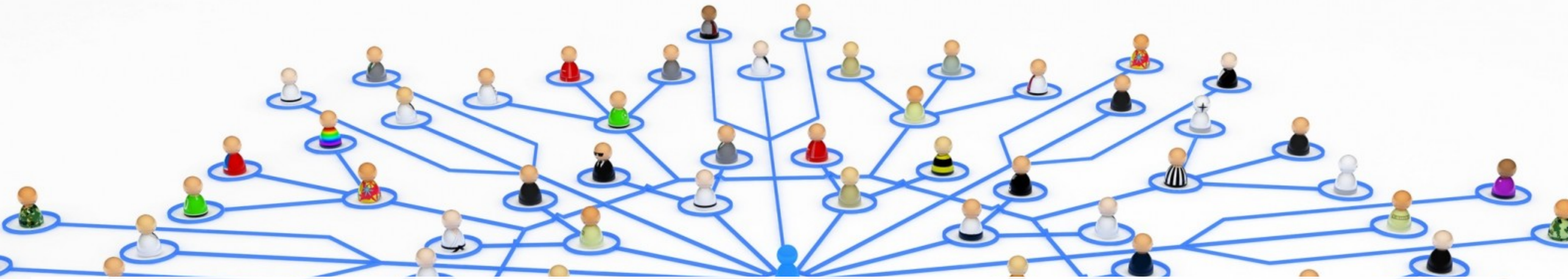
… though they do work great for some things

Responsive Sites

Even as a DS, you want a web view that is live and reactive

It may hit multiple DB’s, load big slow data…

You may want interactivity! You need ***javascript and API’s!***



API's

ORM-as-a-Service

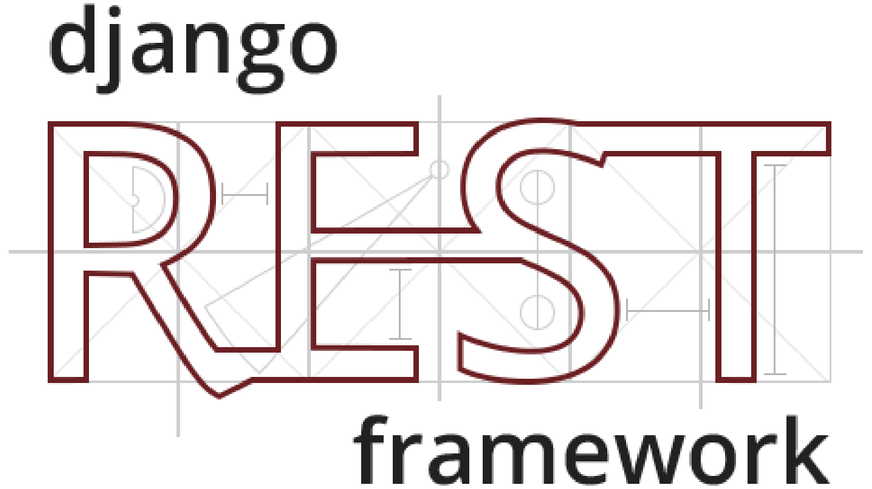
A django view hard-bakes the data into the page

You want to expose your data directly to clients, whether for the web via JS, or any other service (even python)

Django REST Framework

Your first stop to expose your data…

https://www.django-rest-framework.org/



Django REST Framework

It's not really REST, but that's okay

It emphasizes multiple linked endpoints for 'raw' data access

It emphasizes development and debugability

Serializers

It’s easy to expose any Django model…

```
class
AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = '__all__'
```

Serializers

… or any other object

```
class CommentSerializer(serializers.Serializer):
    email = serializers.EmailField()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

Views

… and pack it all into a view

```
class SnippetList(APIView):

    def get(self, request, format=None):
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(
            snippets, many=True)
        return Response(serializer.data)
```

```
    def post(self, request, format=None):
        serializer = SnippetSerializer(
            data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(
                serializer.data,
                status=status.HTTP_201_CREATED)
        return Response(
            serializer.errors,
            status=status.HTTP_400_BAD_REQUEST)
```

Browseable API

Snippet List

OPTIONS GET

This viewset automatically provides `list`, `create`, `retrieve`, `update` and `destroy` actions.

Additionally we also provide an extra `highlight` action.

```
GET /snippets/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
```


```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "url": "https://restframework.herokuapp.com/snippets/1/",
            "id": 1,
            "highlight": "https://restframework.herokuapp.com/snippets/1/highlight/",
            "owner": "admin",
            "title": "serializers.py",
            "code": "from django.contrib.auth.models import User, Group\r\nfrom rest_framework import serializers\r\n\r\n\r\nclass UserSerializer(serializers
            "linenos": false,
            "language": "python",
            "style": "default"
        }
    ]
}
```

Witchcraft

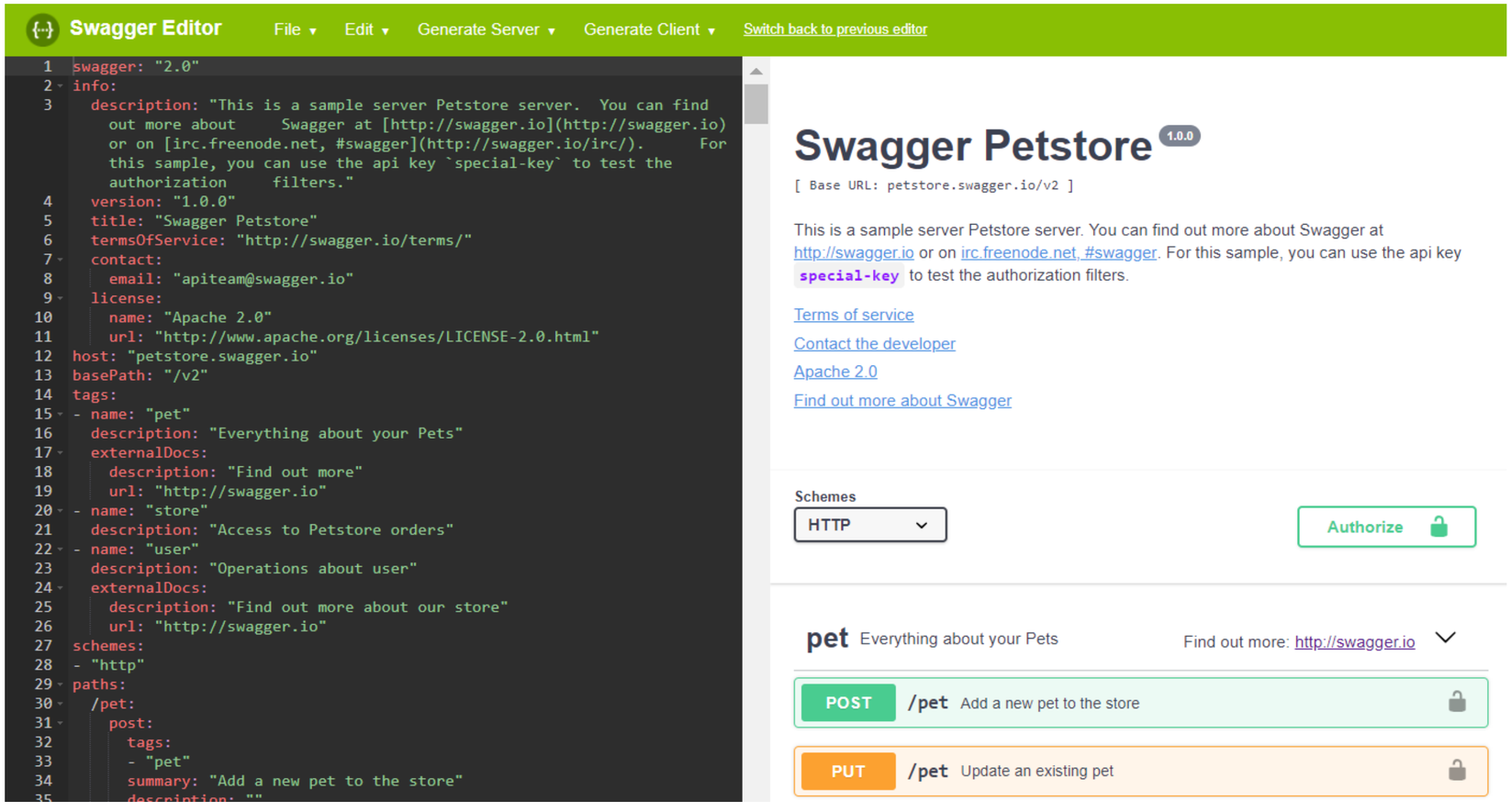
> *django-rest-witchcraft is an extension for Django REST Framework that adds support for SQLAlchemy. It aims to provide a similar development experience to building REST api's with Django REST Framework with Django ORM, except with SQLAlchemy.*
>
> — *Django REST Witchcraft*

Swagger

> *The OpenAPI Specification … is the world's standard for defining RESTful interfaces. The OAS enables developers to design a technology-agnostic API interface that forms the basis of their API development and consumption.*
>
> *Swagger.io*

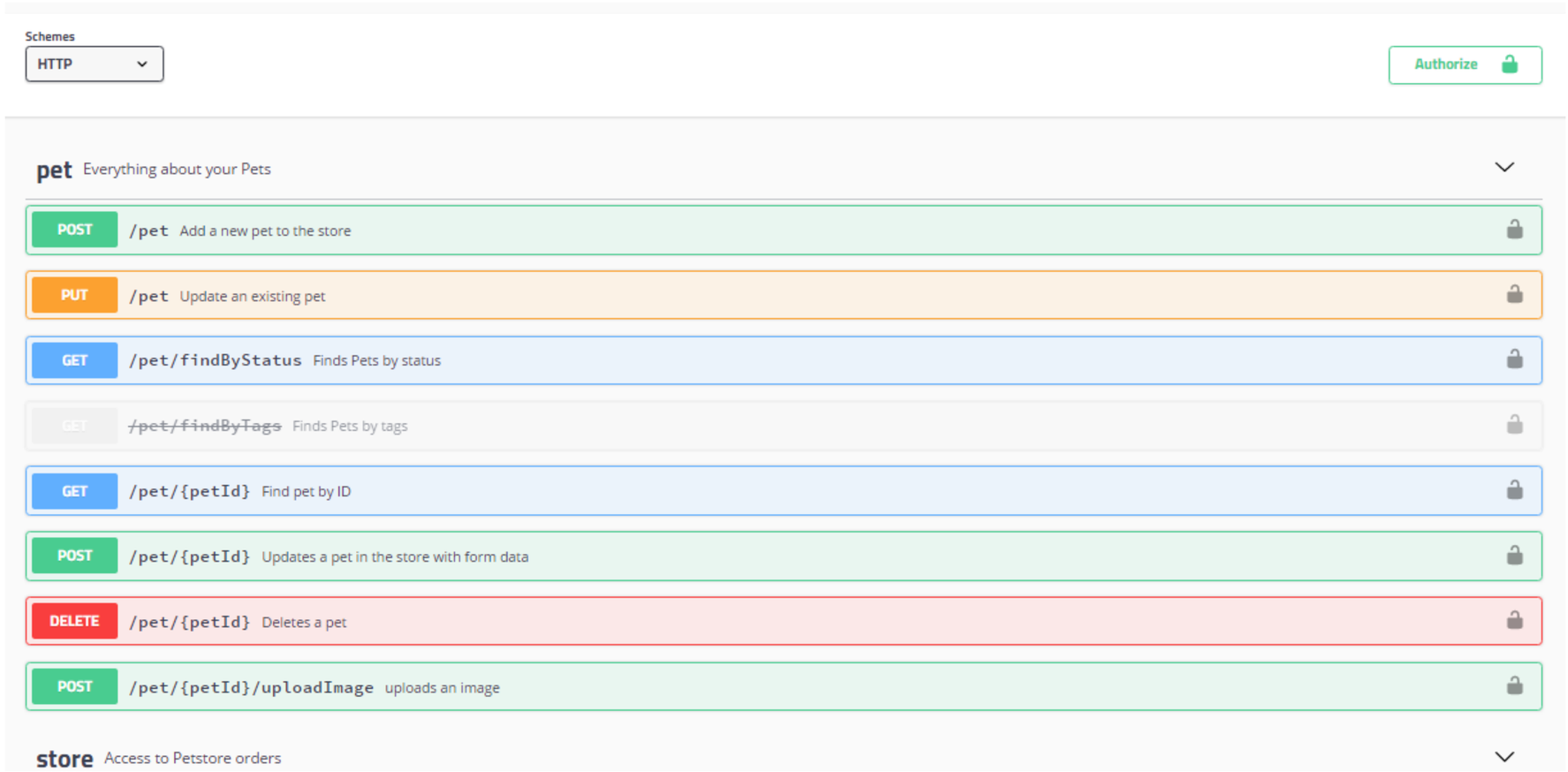
SWAGGER

Editor: Specify API via yaml



Swagger

UI: Browseable API



Swagger

Codegen: Generate client sdk's automatically! (then test with them!)

```
Available Clients: [ akka-scala,
  android, async-scala,clojure,cpprest,csharp,CsharpDotNet2,
  cwiki,dart,dynamic-html,flash,go,groovy,html,
  html2,java,javascript,javascript-closure-angular,
  jaxrs-cxf-client,jmeter,objc,perl,php,python,
  qt5cpp,ruby,scala,swagger,swagger-yaml,swift,
  swift3,tizen,typescript-angular,typescript-angular2,
  typescript-fetch,typescript-node],

Available Servers: [ aspnet5,aspnetcore,
  erlang-server,go-server,haskell,inflector,
  jaxrs,jaxrs-cxf,jaxrs-cxf-cdi,jaxrs-resteasy,
  "jaxrs-spec","lumen","msf4j","nancyfx","nodejs-server",
  python-flask,rails5,scalatra,silex-PHP, sinatra,
  slim,spring,undertow]
~
~
~
~
~
~
~
```

Swagger

The API is data, whether generated from your backend or vice versa. Metacode for client libraries provides consistency and removes boiler plate.

Swagger

Enables new workflows:

- DRF -> generate API spec -> generate client
- Write spec for existing API -> generate client
- Generate server stubs from spec, implement them

Reading

- Python Metaclasses
- Documenting your API
- Browse: Swagger API Design