DJANGO INTRODUCTION

Preliminary

- Target audience
  - Pythoner :p
  - Want to know how Django works
  - Want to construct RESTful API backend based on Django for SPA
- Background knowledge
  - Python (of course)
  - Database fundamental concepts
  - HTTP fundamental concepts

Django

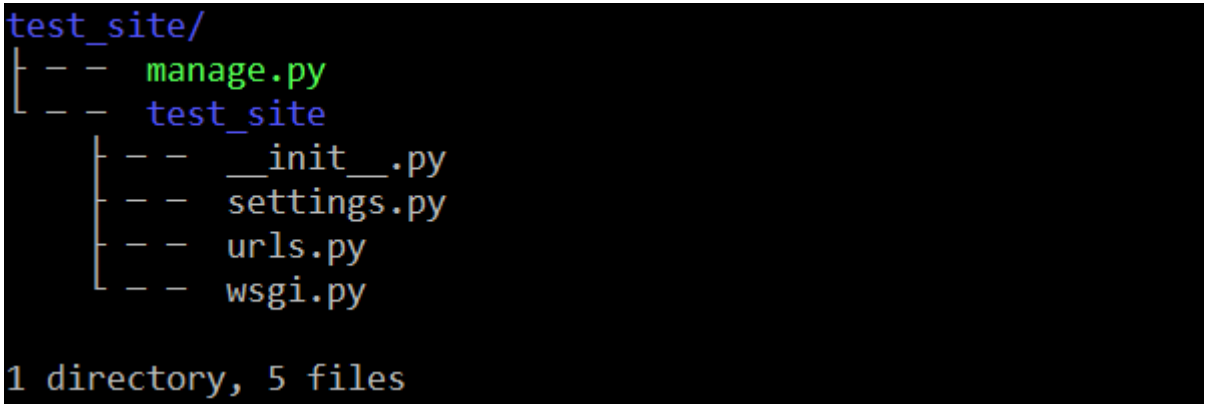
- Web framework
- MVC pattern (Model – View – Controller)
  - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
- (But) Django call itself as MVT pattern
  - Model => Model
  - View => (partial) View + (partial) Controller
  - Template => (partial) View
  - Where is Controller? => Built-in Django framework
- WTX???

MVC vs. MTV

- The point is: the vague boundaries between View & Controller
- Some folks consider (RoR?)
  - View
    - What users see, presentation
  - Controller
    - Determine which data/view should be presented to users
    - Update models according to user's input
- But Django interpret MVC as
  - View
    - Determine which data should be presented to users
    - Delegate presentation (how users see) to Template
  - Controller
    - Dispatch requests to views by URLconf (URL routing)
  - https://goo.gl/YcRj9E

First Step : Start a Project

- # pip install Django
- # django-admin startproject test_site
- Done(!?)
- Directory tree



How Django Works?

- Django's primary deployment platform is WSGI
  - Web server gateway interface
  - https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface
- Then Django (and your Django app) is run as a WSGI application
- Example: Nginx + uWSGI + Django

Nginx + uWSGI + Django

- Concept
  - http://uwsgi-docs.readthedocs.io/en/latest/tutorials/Django_and_nginx.html#concept
- Web client send HTTP request -> web server (Nginx) -> uWSGI -> Django (apps) construct HTTP response -> uWSGI -> web server (Nginx) -> web client
- The entry point of Django is the “application” (callable) object in wsgi.py under the project (Ex. test_site/wsgi.py)
  - The “application” object follows WSGI spec.

A little bit Deep Dive

- class WSGIHandler & WSGIHandler's method "__call__" in Django's source code
  - core/handlers/wsgi.py
  - core/handlers/base.py
- It shows the "whole" control flow of Django apps
  1. "__call__" is called when there is a coming HTTP request through WSGI
  2. Run "request middleware" with the request, goto while there is a response
  3. Run "view middleware" with the request, goto while there is a response
  4. Run your app with the request, got while there is a response
  5. Run "template middleware" if it's necessary for the response
  6. Run "response middleware" with the request & response
  7. Return response

advanced

Django Control Flow

- Receive a HTTP request
- Dispatch a request to a “View” according to URL configurations (test_site/urls.py)
- Construct the HTTP response in View
- Return the HTTP response

Django Control Flow : View

- It has a high probability that the most of operations to get data from model in a view are database accesses
  - So, Django provides its own ORM (“Model”)
- Finally, a response should be generated according the data “Model”. Django provides its own “Template” engine for convenience
- Follow-up
  - View
  - Model
  - Skip “Template” because it’s meaningless in SPA

VIEW

URL Configuration & View

- Dispatch HTTP request to Views according to URL configurations
  - ROOT_URLCONF = ‘test_site/urls’ (in test_site/settings.py)
  - test_site/urls.py

URL Configuration

- (Functional) View
  - hello(), contact(), product() are function objects
- url(pattern, view [, other_kwargs])
  - Pattern: use python “re” library
    - https://docs.python.org/2/library/re.html

```
urlpatterns = [
    url('^hello/$', hello),
    url('^contact/$', contact),
]

urlpatterns += [
    url('^product/$', product),
]
```

URL Configuration

- url() with arguments
  - Group concept in “re” library
  - Group => positional arguments
  - Named group => keyword arguments

```
urlpatterns = [
    url('^user/($\w+)/$', user),
    # positional argument
    # URL: user/foobar/ => user('foobar')
]

urlpatterns = [
    url('^user/(?P<name>\w+)/(?P<date>[0-9]+)/$', user),
    # keyword argument
    # URL: user/foobar/20170101/ => user(name='foobar', date='20170101')
]
```

URL Configuration

- url() with extra argument (optional)
  - Must be a dictionary
  - Will override arguments defined by group results if there are conflicts

```
urlpatterns = [
    url('^user/(?P<name>\w+)/20170101/$', user, {'date': '20170101', 'type': 'normal'}),
    # URL: user/foobar/20170101/ => user(name='foobar', date='20170101', type='normal')

    url('^user/(?P<name>\w+)/(?P<date>[0-9]+)/$', user, {'date': '20170101', 'type': 'normal'}),
    # URL: user/foobar/20161231/ => user(name='foobar', date='20170101', type='normal')
]
```

URL Configuration

- include() other URL configuration files / settings
  - Arguments defined in the upper levels are all passed in

```
urlpatterns = [
    url('^user/(?P<name>\w+)/profile/$', user.profile, {'type': 'normal'}),
    url('^user/(?P<name>\w+)/record/$', user.record, {'type': 'normal'}),
    url('^user/(?P<name>\w+)/mail/$', user.mail, {'type': 'normal'}),
]

urlpatterns = [
    url('^user/(?P<name>\w+)/', include(user_urls), {'type': 'normal'}),
]

# user_urls.py
urlpatterns = [
    url('^profile/$', user.profile),
    url('^record/$', user.record),
    url('^mail/$', user.mail),
    # URL: user/foobar/profile/ => user.profile(name='foobar', type='normal')
]
```

Class-Based View

- Reuse code (DRY) => OO (sigh…)
- Derive from “View” class
- HTTP request types as its method names (lower cases)

```
from django.views import View

class UserView(View):
    def get(self, request, *args, **kwargs):
        # handle GET
        pass

    def post(self, request, *args, **kwargs):
        # handle POST
        pass

    def put(self, request, *args, **kwargs):
        # handle PUT
        pass

    ...

urlpatterns = [
    url('^user/(?P<name>\w+)/$', UserView.as_view(), {'type': 'normal'}),
]
```

Class-Based View

- UserView.as_view() returns a function bound with an instance of UserView (closure)
- So, it can be mixed with a lot of existed views
  - By multiple inheritances (sigh…)
  - Mixin (err… think it as Java's interface + implementation)
- Anyway, the documents s*cks
  - Read the source code (sigh…)
- https://docs.djangoproject.com/en/1.8/ref/class-based-views/

advanced

View

- Responsibility
  - Return a HTTP response
  - That’s it!

```
def user_view(request):
    html = "<html><body>Hello World!</body></html>"
    return HttpResponse(html)
```

MODEL

Before Write Models

- Carefully design database tables
  - Django's model is targeted on relational database
- Database normalization
  - https://en.wikipedia.org/wiki/Database_normalization
  - 1NF, 2NF, 3NF
- May need de-normalization for performance considerations
- General principle
  - Design normalized database first (1NF, 2NF, 3NF)
  - De-normalize ONLY if necessary

Model Configuration

- settings.py
  - Set DATABASE with ENGINE, NAME, USER, PASSWORD, HOST, PORT
  - Make sure your app is in INSTALLED_APPS
- Models are existed in your_app/models.py or import models in your_app/models/__init__.py (utilize package stuff)

Model

- Django has own ORM for database accesses
- Model
  - All models should derive from models.Model
  - Class variable => table fields

```
from django.db import models

class User(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

Primary Key

- If there is no argument “primary_key=True” for any fields in a model, Django will automatically add an model.AutoField() named id

```
class User(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    # implicit
    id = models.AutoField(primary_key=True)

# specify a primary key
class User(models.Model):
    first_name = models.CharField(max_length=30, primary_key=True)
    last_name = models.CharField(max_length=30)
```

Field Types

https://docs.djangoproject.com/en/1.8/ref/models/fields/#model-field-types

- AutoField
  - An IntegerField that automatically increments according to available IDs
- BinaryField
- BooleanField
- CharField
- DateField
- FileField
- FloadField
- ImageField
- IntegerField
- UUIDField
- …

Field Types

- Relationship fields
- ForeignKey
- ManyToManyField
- OneToOneField

Model Inheritance

- Reuse code (DRY) => OO (again…)
- Inheritance style
  - No model inheritance (LOL)
  - Abstract base classes
  - Multi-table inheritance
  - Proxy models

Abstract Base Classes

- NOT the abstract class in python
- A abstract base class cannot be used as a model
- Every derived class will create its own table (only one table) with the fields defined in its abstract base class

```
class CommonInfo(models.Model):
    age = models.IntegerField()
    name = models.CharField(max_length=60)

    class Meta:
        abstract = True

class User(CommonInfo):
    group = models.CharField(max_length=60)
```

Multi-table inheritance

- Tables are created for both parent and child
- Automatically created OneToOne field links parent & child
- DO NOT use this style !!!

```
class Place(models.Model):
    address = models.CharField(max_length=100)
    name = models.CharField(max_length=60)

class Shop(Place):
    dress = models.BooleanField(default=False)
    shoes = models.BooleanField(default=False)
    food = models.BooleanField(default=False)

p = Place.objects.get(id=2) # if p is a instance of Shop
p.shop # access Shop's members
```

Proxy Models

- All the operations on data are saved to the original model (base model)
- Can apply different behaviors in proxy models: different ordering, model manager and etc.

```
class User(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class GoldUser(User):
    class Meta:
        proxy = True

    def op(self):
        pass
```

QuerySet

- https://docs.djangoproject.com/en/1.8/ref/models/querysets/
- Query database models
  - User.objects is a model manager (discuss this later)
  - User.objects.all() returns a QuerySet which contains all records
  - User.objects.filter() returns a QuerySet which contains filtered records
  - User.objects.exclude() returns a QuerySet which contains records without matched conditions
  - User.objects.get() returns the (only one) matched instance of model

Query Arguments

- UserModel.objects.filter(**kargs)
  - kargs: field_name1=value1, field_name2=value2, …
  - field_name1=value1 AND field_name2=value2 AND …
- .exclude() == complement of .filter()
- UserModel.objects.get(**kargs) will raise exceptions when there are more than one results

```
User.objects.filter(first_name='Foo', last_name='Bar')
# Users named Foo Bar, return a QuerySet

User.objects.exclude(first_name='Foo', last_name='Bar')
# Users not named Foo Bar, return a QuerySet

User.objects.get(first_name='Foo', last_name='Bar')
# User named Foo Bar, there must be only one Foo Bar
# Return an instance of User model
```

Query Arguments

- More complex queries
  - https://docs.djangoproject.com/en/1.8/ref/models/querysets/#field-lookups
- Field lookups: suffixed field_name with __OP (OP prefixed with two underscores)

```
User.objects.filter(last_name='Bar', age__gt=20)
# Mr. Bar(s) with age greater than 20

User.objects.filter(first_name='Foo', age__in=[10,20,30])
# Foo(s) with age 10 or 20 or 30
```

Complex Lookups

- Use django.db.models.Q to construct complex lookups (like OR)
- & (AND), | (OR), ~(NOT)
- Can co-exist with basic lookups

```
from django.db.models import Q

User.objects.filter((Q(age__gt=65) | Q(age__lt=15)) & ~Q(first_name='Foo'))
# equal to
User.objects.filter(Q(age__gt=65) | Q(age__lt=15), ~Q(first_name='Foo'))

User.objects.filter(Q(age__gt=65) | Q(age__lt=15), first_name='Foo')
# Q() is a positional argument, first_name='Foo' is a keyword argument
# MUST put positional arguments before keyword arguments
```

Query Ordering

- Default ordering can be specified in the Meta of model classes.
- UserModel.objects.order_by(‘-field_1’, ‘field_2’)
  - Default is ascending
  - ‘-’ means descending
  - Ordered by field_1 first, then by field_2

```
class User(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    age = models.IntegerField()

    class Meta:
        ordering = ['last_name', '-age']

# or use order_by()
User.objects.order_by('last_name', '-age')
```

Create

By Model

1. New an instance of the model
2. (Optional) modify the object
3. obj.save()

By QuerySet

- User.objects.create(**kargs)
- Create & save in on step

```
u = User(first_name='Foo', last_name='Bar')
u.save()


u = User.objects.create(first_name='Foo', last_name='Bar')

# is equivalent to

u = User(first_name='Foo', last_name='Bar')
u.save(force_insert=True)
```

Read

- Use query related methods
  - .all(), .filter(), .exclude(), .get() …

Update

By Model

1. Get the target object
2. Modify the object
3. obj.save()

By QuerySet

- User.objects.filter(**kargs)
  .update(**kargs)

```
u = User.objects.get(first_name='Foo')
u.age = 30
u.save()


User.objects.filter(first_name='Foo').update(age=30)
```

Delete

By Model

1. Get the target object
2. obj.delete()

By QuerySet

- User.objects.filter(**kargs)
  .delete()

```
u = User.objects.get(first_name='Foo')
u.delete()

User.objects.filter(first_name='Foo').delete()
```

CRUD via Model or QuerySet

- The CRUD operations via QuerySet methods directly issue SQL commands to database. It won't call any method of the target model. No signal of the model would be emitted.
- The CRUD operations via Model methods will execute its codes + SQL commands

Access ForeignKey

- The member is an instance of referred model
- Reverse access
  - Lowercase of model name + “__set” (two underscores)
  - It’s a QuerySet

Access ForeignKey

```
class Group(models.Model):
    name = models.CharField(max_length=60)

class User(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    age = models.IntegerField()
    group = models.ForeignKey('Group')

u = User.objects.get(id=10)
u.group # an instance of Group

g = Group.objects.get(name='foobar')

# reverse access
# g.user_set is a QuerySet which contains users of group foobar
g.user_set.all()
g.user_set.filter(age__gt=20)
```

Access ManyToManyField

- The member is a QuerySet of referred model
- Reverse access: the same of ForeignKey

```
class Group(models.Model):
    name = models.CharField(max_length=60)

class User(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    age = models.IntegerField()
    group = models.ManyToManyField('Group')

u = User.objects.get(id=10)
u.group # a QuerySet for Group
u.group.filter(name__contains='foo')

g = Group.objects.get(name='foobar')

# reverse access
# g.user_set is a QuerySet which contains users of group foobar
g.user_set.all()
g.user_set.filter(age__gt=20)
```

Access OneToOneField

- The member is an instance of referred model
- Reverse access
  - Lowercase of model name
  - It’s an instance of the model

Access OneToOneField

```
class Group(models.Model):
    name = models.CharField(max_length=60)

class User(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    age = models.IntegerField()
    group = models.OneToOneField('Group')

u = User.objects.get(id=10)
u.group # an instance of Group

g = Group.objects.get(name='foobar')

# reverse access
# g.user is an instance of User
g.user
```

Model Manager

- UserModel.objects is a model manager
- Performs database queries
- Customized
  - Derive from models.Manager
  - Or use QuerySet.as_manager()

advanced

Model Manager

```
class CustomManager(models.Manager):
    def custom_method(self):
        pass

class User(models.Model):
    custom_mgr = CustomManager()

User.custom_mgr.custom_method()


class CustomQuerySet(models.QuerySet):
    def custom_method(self):
        pass

class User(models.Model):
    custom_mgr = CustomQuerySet.as_manager()

User.custom_mgr.custom_method()
```

advanced

Perform RAW Queries

- Use method .raw() of model manager
  - User.objects.raw(query_string)
- Use database connection & cursor

```
User.objects.raw('SELECT * FROM User')

# Custom Model Manager
from django.db import connection

class CustomManager(models.Manager):
    def get_user(self):
        cursor = connection.cursor()
        cursor.execute('SELECT * FROM User')
        for row in cursor.fetchall():
            pass
        return result
```

advanced

SIGNAL

Signal

- Execute receiver function when signal is emitted
- Built-in signals
  - https://docs.djangoproject.com/en/1.8/ref/signals/
- Listen signals
  - Signal.connect(recv_callback, sender=cls_sender)

```
from django.db.models.signals import pre_save

def callback(sender, **kargs):
    pass

pre_save.connect(callback, sender=User)
```

advanced

Customize Signal

- All signals are django.dispatch.Signal instances
- Create:
  - signal = django.dispatch.Signal(providing_args=[arg1, arg2])
- Send (emit)
  - signal.send(sender=cls_sender, arg1=arg1, arg2=arg2)
  - Return a list of (receiver, response)
- .send() vs .send_robust()
  - send() doesn’t catch any exceptions raised by receivers

advanced

SESSION

Session Middleware

- Enable session middleware in settings.py
  - MIDDLEWARE_CLASS has ‘django.contrib.sessions.middleware.SessionMiddleware’
- Database-backed sessions
  - INSTALLED_APPS has ‘django.contrib.sessions’
- Other types: set SESSION_ENGINE
  - django.contrib.sessions.backends.cache
  - django.contrib.sessions.backends.file
  - django.contrib.sessions.backends.signed_cookies

Session Middleware

- request.session object, it can be used as dictionary
  - request.session[‘var’] = value
  - request.session.get(‘var’, default)
- Other methods: related to session behaviors
  - set_test_cookie()
  - test_cookie_worked()
  - delete_test_cookie()
  - set_expiry(*value*)
  - get_expiry_age()
  - get_expiry_date()
  - (https://docs.djangoproject.com/en/1.8/topics/http/sessions/#using-sessions-in-views)

ADMIN SITE

Admin Interface

- URL configuration
  - url('^admin/', include(admin.site.urls)),
- Add your models
  - Add "admin.py" in the application's directory
  - admin.site.register(model_name, [ModelAdmin])
- Model's methods & attributes related to Admin
  - __str__(): the representation of entries
  - get_absolute_url(): switch between the admin view and the object's detail view
  - (class) Meta.ordering: display ordering
  - Meta.verbose_name: display name

Customized Admin Interface

- Derive from admin.ModelAdmin
- The second argument of admin.site.register(model_name, ModelAdmin)
- https://docs.djangoproject.com/en/1.8/ref/contrib/admin/

DJANGO REST FRAMEWORK

Django REST Framework

- One of the most popular framework/library for building REST APIs
  - http://www.django-rest-framework.org/
- How does it work?
  1. Receive HTTP requests
  2. Dispatch the request to a view
  3. Get/Construct objects of target model
  4. Serialize the objects to JSON/browsable page as HTTP response
  5. Return the HTTP response
- View + (Model) Serialization

Serializer

- Derive from rest_framework.serializers
- Need to implement two methods
  - create(self, validated_data)
    - validated_data: a dictionary contains ‘field_name’: value pairs
  - update(self, instance, validated_data)
    - Instance: instance of target model (Django ORM)

Serializer

```python
from rest_framework import serializers

class User(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    age = models.IntegerField()

class UserSerializer(serializers.Serializer):
    first_name = serializers.CharField()
    last_name = serializers.CharField()
    age = serializers.IntegerField()

    def create(self, validated_data):
        return User.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.first_name = validated_data.get('first_name', instance.first_name)
        instance.last_name = validated_data.get('last_name', instance.last_name)
        instance.age = validated_data.get('age', instance.age)
        instance.save()
        return instance
```

Serializer

- Serializer fields
  - http://www.django-rest-framework.org/api-guide/fields/
- Two direction behaviors
  - GET => Model => serialization => JSON (or other format)
  - PUT or POST => de-serialization => save to model

Serialization (GET)

- Steps:
  1. Get an instance of a model
  2. New a serializer’s object of corresponding serializer class
  3. serializer.data
  4. JSONRender().render(serializer.data) or use “Response”
- Use QuerySet instead of model instance: new serializer’s object with argument “many=True”

Serialization (GET)

```python
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser

def user_view(request):
    if request.method == 'GET':
        u = User.objects.get(first_name='Foo')
        serializer = UserSerializer(u)
        serializer.data # serialized data
        content = JSONRenderer().render(serializer.data)
        return HttpResponse(content, content_type='application/json')

def user_list_view(request):
    if request.method == 'GET':
        users = User.objects.all()
        serializer = UserSerializer(users, many=True)
        serializer.data # serialized data
        content = JSONRenderer().render(serializer.data)
        return HttpResponse(content, content_type='application/json')
```

De-serialization (POST)

- Step:
  1. JSONParser().parse(stream) => “data”
  2. New serializer’s object with argument “data=data_get_by_step1”
  3. serializer.is_valid() , can get result by serializer.validated_data
  4. serializer.save()
- .save() will call .create() or .update() implementation

De-serialization (POST)

```python
def user_view(request):
    if request.method == 'POST':
        data = JSONParser.parse(request)
        serializer = UserSerializer(data=data)
        if serializer.is_valid():
            serializer.save() # save to model
            content = JSONRenderer().render(serializer.data)
            return HttpResponse(content, content_type='application/json', status=201)
        else:
            content = JSONRenderer().render(serializer.errors)
            return HttpResponse(content, content_type='application/json', status=400)
```

ModelSerializer

- Define mapping model name & available fields in sub-class Meta

```
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ('first_name', 'last_name', 'age')
```

(Wrapped) Request & Response

- Use @api_view([method_list]) for functional view, derive from APIView for class-based view
- Request
  - request.method: HTTP method name
  - request.data: POST/PUT data
- Response
  - serializer.data as argument
  - Return JSON as HTTP response
- format_suffix_patterns
  - Provide different rendering format
  - Browsable API
  - Additional parameter “format” for views

Request & Response

```python
@api_view(['GET', 'POST'])
def user_list_view(request):
    if request.method == 'GET':
        users = User.objects.all()
        serializer = UserSerializer(users, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = UserSerializer(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)

# url.py
from rest_framework.urlpatterns import format_suffix_patterns

urlpatterns = [
    url(r'^users/$', user_list_view),
]
urlpatterns = format_suffix_patterns(urlpatterns)
```

Class-Based View

- Mapping method names to HTTP methods
  - .get(), .post(), .put(), .delete() …

```
class UserList(APIView):
    def get(self, request, format=None):
        users = User.objects.all()
        serializer = UserSerializer(users, many=True)
        return Response(serializer.data)

    def post(self, request, format=None):
        serializer = UserSerializer(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)
```

Class-Based View

- Mixin (again…)
- Generic class-based views (endless wrapper…)
  - http://www.django-rest-framework.org/api-guide/generic-views/
- (Do you think it's the end. No, it's the beginning…)
- ViewSet & Router
  - http://www.django-rest-framework.org/tutorial/6-viewsets-and-routers/
- Trade-offs between View & ViewSet
  - Explicit vs Implicit

advanced

TODO

- Authentication
- Security
- Cache framework