Intro to Serverside API Development Using Django

Building a Fortress in a greenfield

Dr. Hale

University of Nebraska at Omaha

Secure Web Application Development – Lecture 3

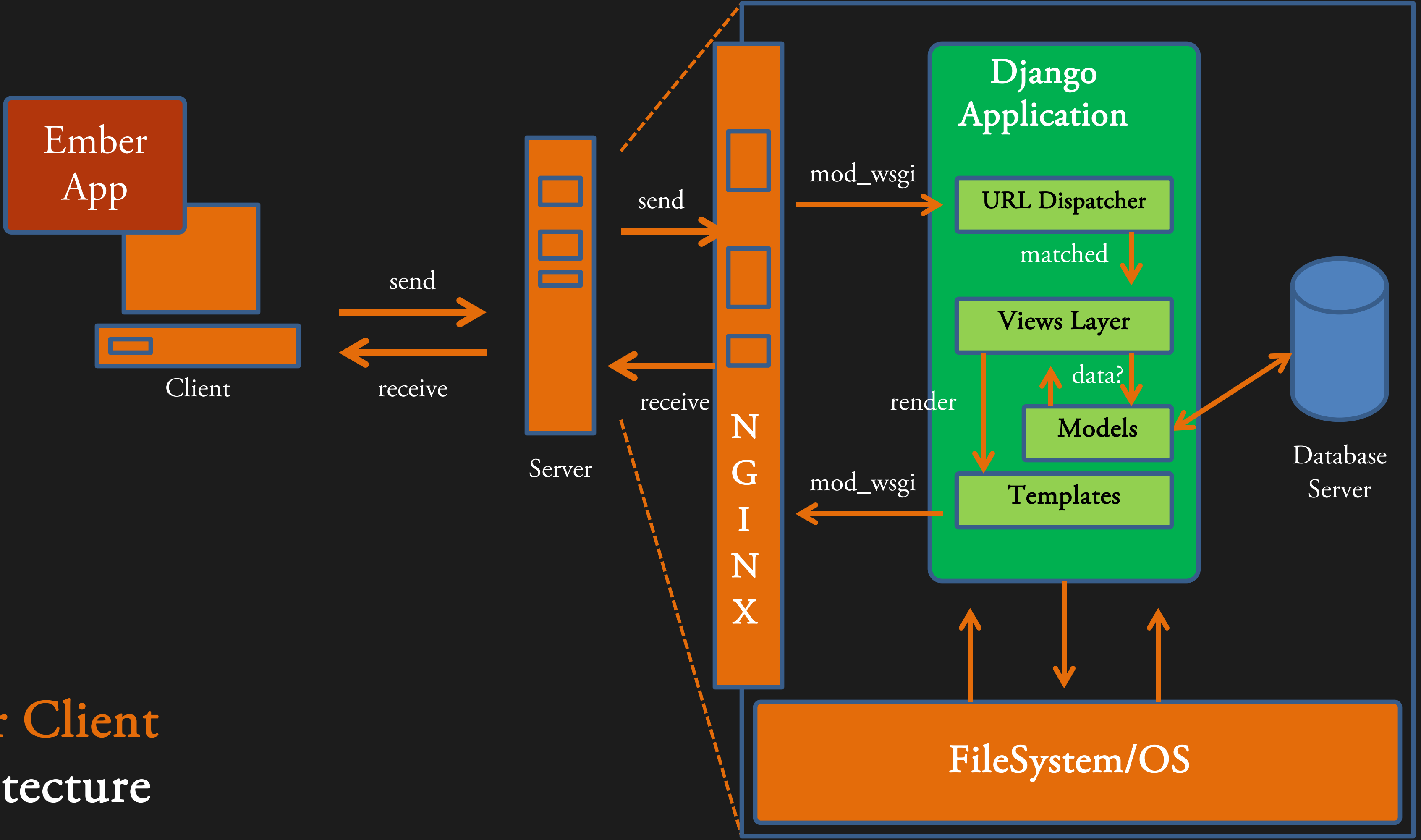
Today's topics:

Server-client

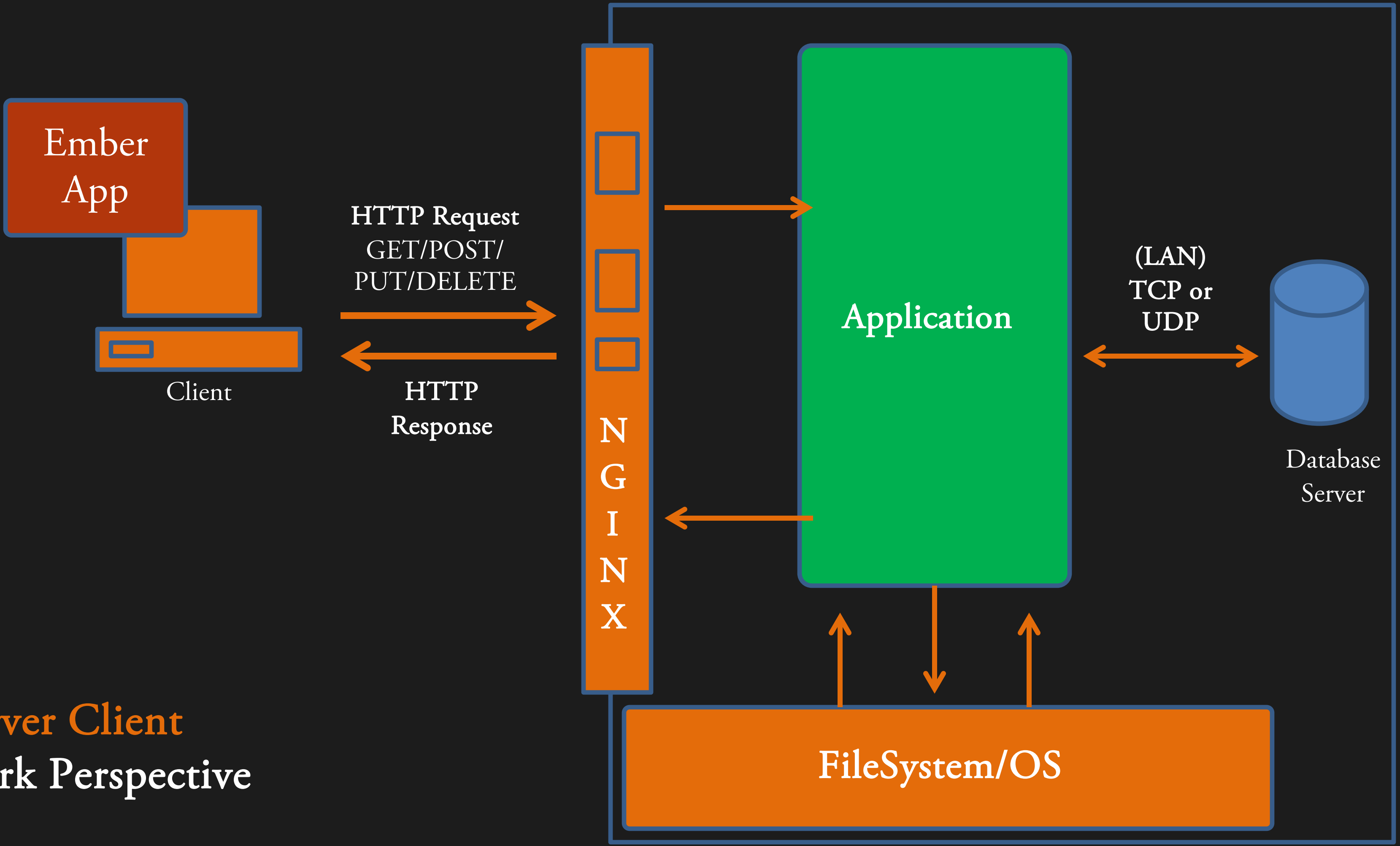
- Architecture and dataflow
- Network Perspective (overview)
- Attack Vectors: Types and where they occur

Django overview

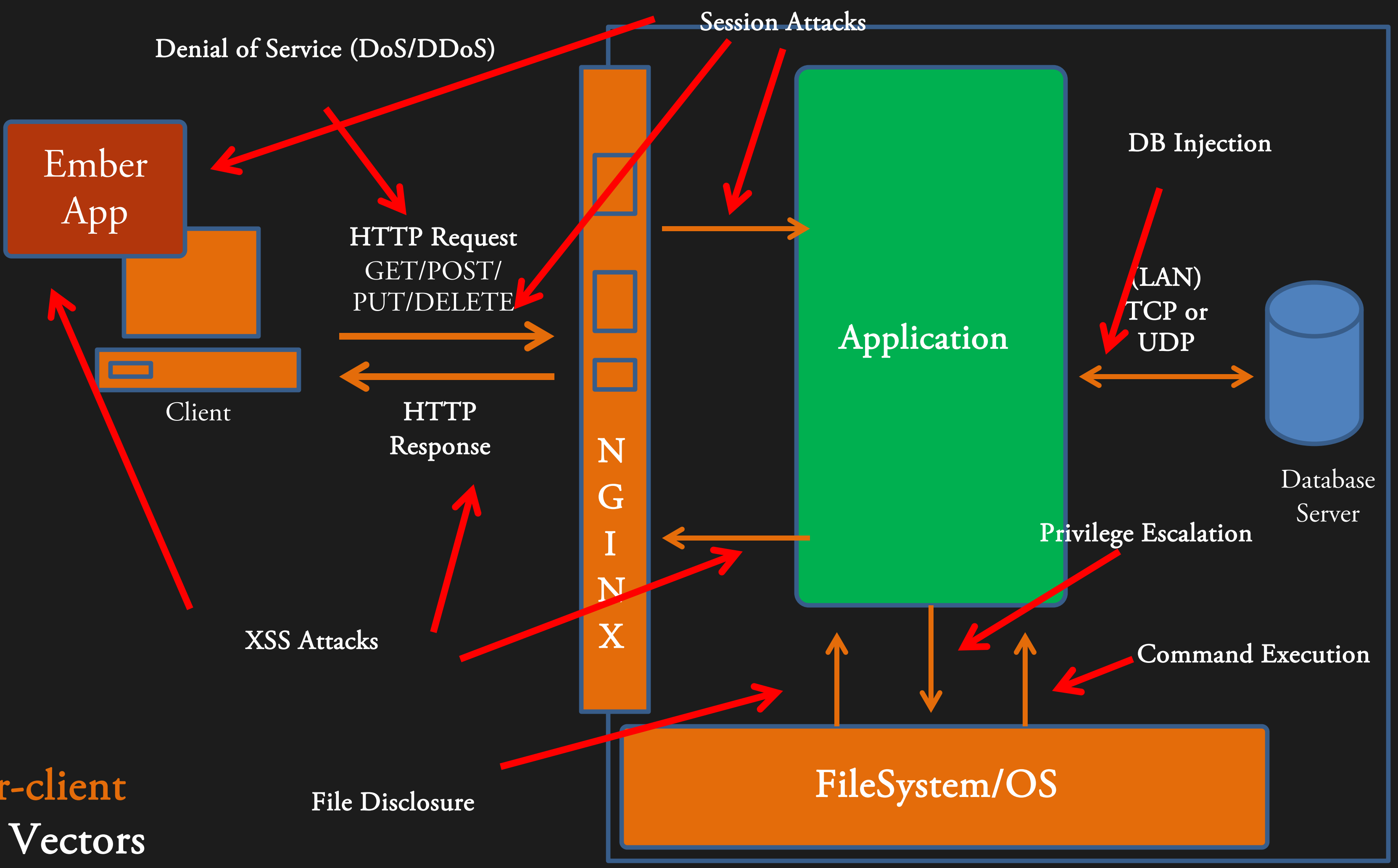
- Intro to Django
- Your application architecture
- Building an API



Server Client Architecture



Server Client
Network Perspective



Server-client
Attack Vectors

We will talk more about defending against these attacks moving forward and you will mitigate them by hardening the API (later) and apache (next)

Web Servers

Apache / Nginx
We will come back to this

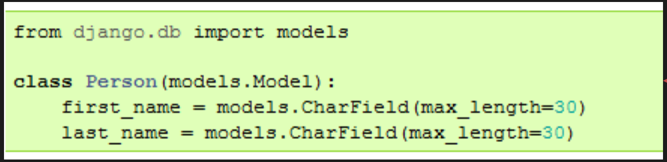
Apache/ NGINX is just the http server.
What about the web framework?

Enter: Django

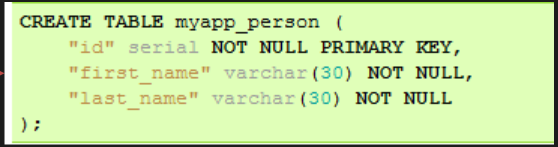
- A high-level web framework
- Automates key web development patterns
- Provides an infrastructure so you can focus on keeping code clean and efficient
- Model-View-Controller pattern, keep it separate!
  - Model (describes database table)
  - Views (handles exchange between user and database, business logic, bad name – these are actually the controllers in django)
  - URLs (map a URL pattern to particular view, similar to an ember route)
  - Templates (specifies presentation format, these are basically the 'view' layer)

Django: Models

- Model ⇔ Database Table
- Model Instance ⇔ Database Record
- Database-abstraction API via object-relational mapping (ORM)
- Helps avoid boilerplate database code
  - e.g. MySQLdb.connect(params=values)



```python
from django.db import models

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

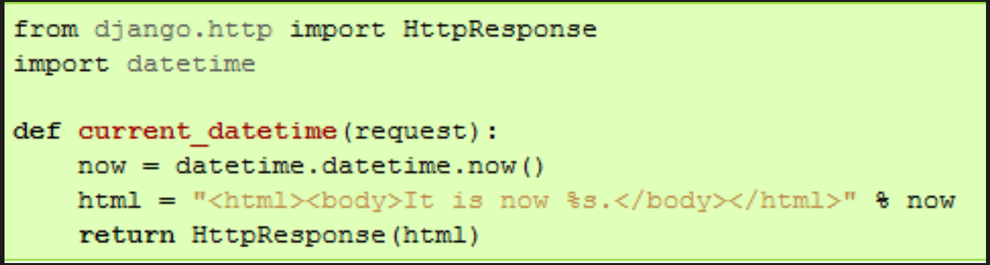


```sql
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

See django model documentation:
https://docs.djangoproject.com/en/1.9/topics/db/models/

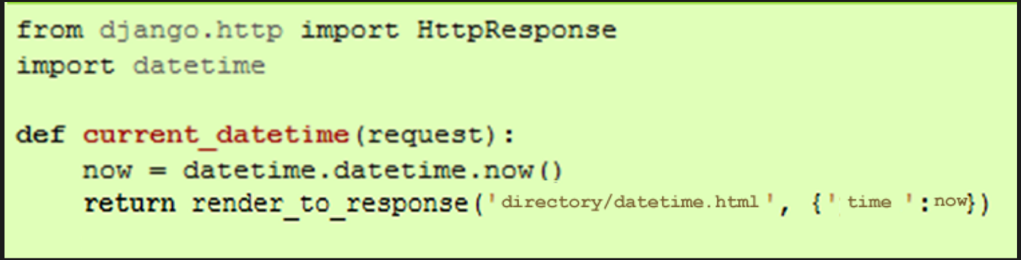
Django: Views (remember these are controllers)

- A simple View:



```
from django.http import HttpResponse
import datetime

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

- An alternate view, utilizing the Django template system:

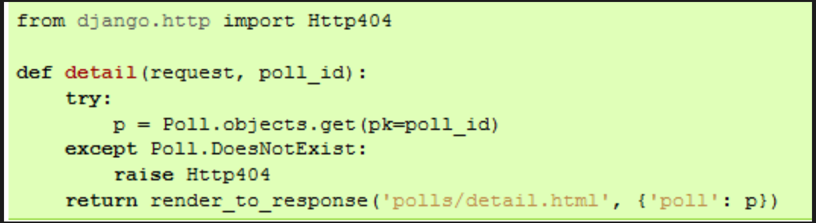


```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('directory/datetime.html', {'time':now})
```

See django view documentation:
https://docs.djangoproject.com/en/1.9/topics/http/views/

Django: Views and simple queries

- Accessing an object and raising a 404 if it doesn't exist



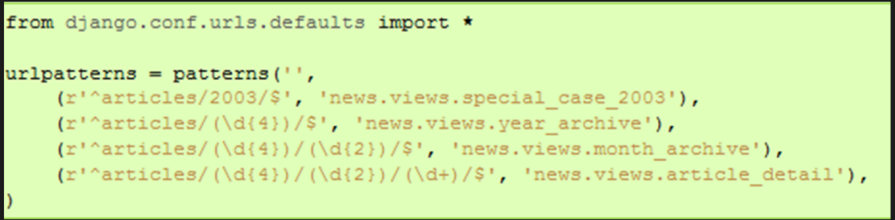
```
from django.http import Http404

def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render_to_response('polls/detail.html', {'poll': p})
```

- Uses some model named "Poll" using the "get" query with a primary key "pk" = poll_id
  - Note: "get" returns one item, use "filter" for sets of items
- Where does poll_id come from? - urls

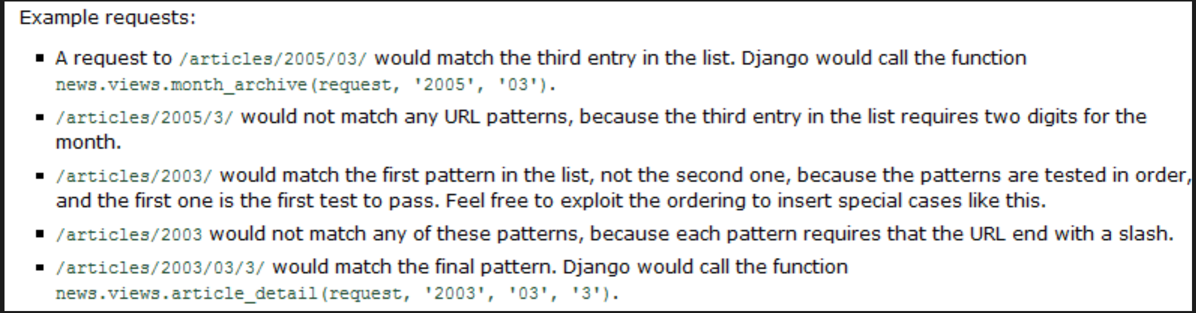
See django view documentation:
https://docs.djangoproject.com/en/1.9/topics/http/views/

Django: URLconf

- The 'Table of Contents' of your web site
  - Mapping between URL patterns and view functions to handle URLs
    - Regular expressions used to specify patterns (don't be afraid if you don't know regex though)



```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^articles/2003/$', 'news.views.special_case_2003'),
    (r'^articles/(\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'news.views.month_archive'),
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'news.views.article_detail'),
)
```

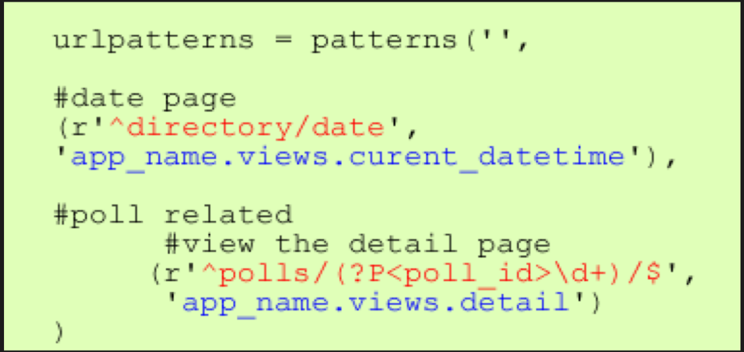


Example requests:

- A request to `/articles/2005/03/` would match the third entry in the list. Django would call the function `news.views.month_archive(request, '2005', '03')`.
- `/articles/2005/3/` would not match any URL patterns, because the third entry in the list requires two digits for the month.
- `/articles/2003/` would match the first pattern in the list, not the second one, because the patterns are tested in order, and the first one is the first test to pass. Feel free to exploit the ordering to insert special cases like this.
- `/articles/2003` would not match any of these patterns, because each pattern requires that the URL end with a slash.
- `/articles/2003/03/3/` would match the final pattern. Django would call the function `news.views.article_detail(request, '2003', '03', '3')`.

See django url documentation:
https://docs.djangoproject.com/en/1.9/topics/http/urls/

Django: The poll detail example

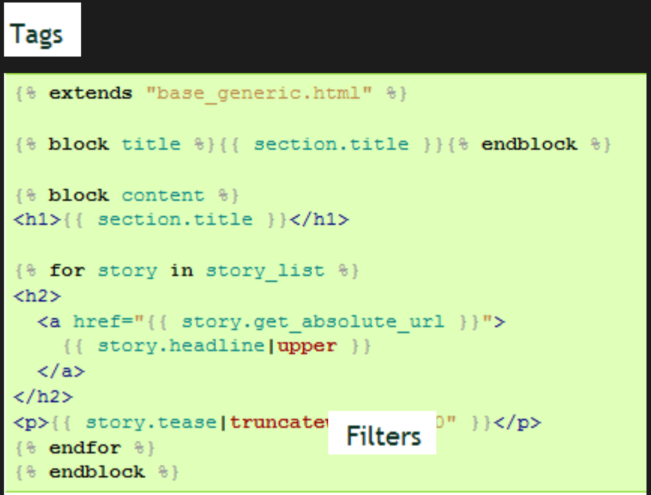


```
urlpatterns = patterns('',

#date page
(r'^directory/date',
'app_name.views.curent_datetime'),

#poll related
      #view the detail page
     (r'^polls/(?P<poll_id>\d+)/$',
      'app_name.views.detail')
)
```

- A request comes in for URL /app_name/polls/detail/12
- Search URLconf for pattern
- Match second pattern, send to app_name.views.detail view function
- Passes HttpRequest object and poll_id represented by one or more digits
- View performs business logic and returns an HttpResponse object

That's great! But what does a template look like?

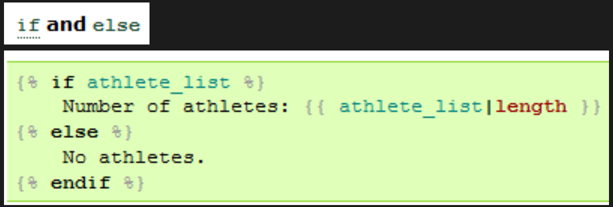
- Templates
  - Placeholder variables
  - Basic logic (template tags)
  - Formatting variables (filters)



Tags

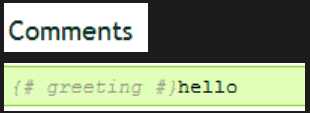
```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncate      0" }}</p>
{% endfor %}
{% endblock %}
```

Filters



if and else

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% else %}
    No athletes.
{% endif %}
```



Comments

```
{# greeting #}hello
```

See django template documentation:
https://docs.djangoproject.com/en/1.9/topics/templates/

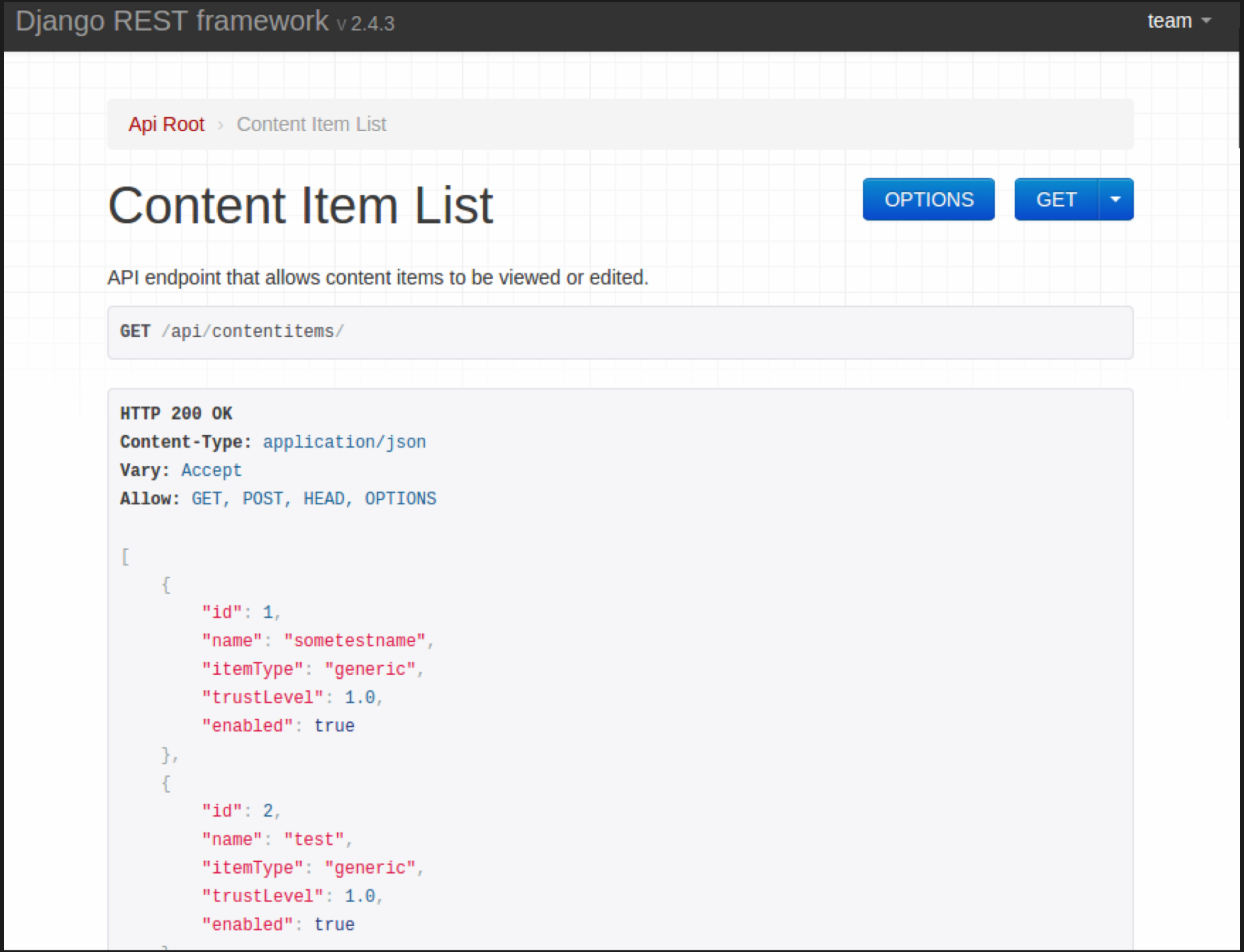
Since your apps are built in the client-side (ember) you are just using the API (next) – so you probably wont need django templates

Django: Bonus

- Admin interface
- Django Packages: Reusable apps, tools and more
  - If you can think of something its probably already been done
  - Use and re-use libraries – don't reinvent the wheel if you don't need to
  - Very similar community to Ember addons
    (but actually even more mature)

Building a REST API in Django

Django REST Framework



Content Item List

API endpoint that allows content items to be viewed or edited.

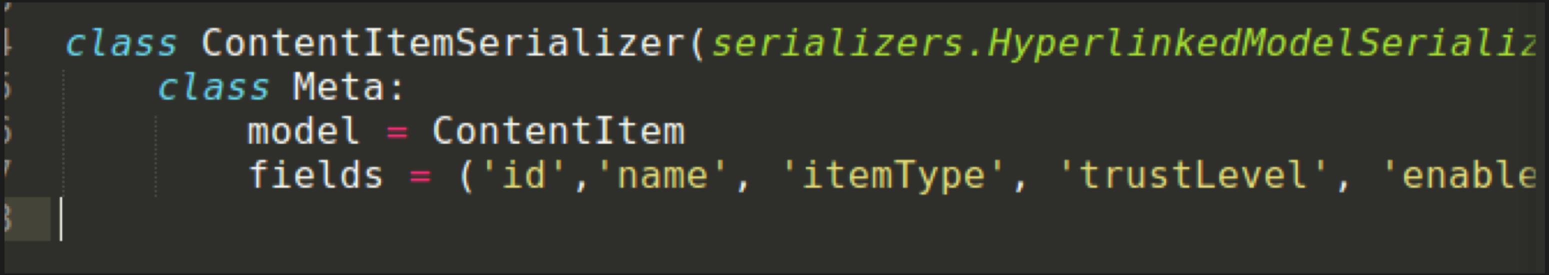
```
GET /api/contentitems/
```

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

[
    {
        "id": 1,
        "name": "sometestname",
        "itemType": "generic",
        "trustLevel": 1.0,
        "enabled": true
    },
    {
        "id": 2,
        "name": "test",
        "itemType": "generic",
        "trustLevel": 1.0,
        "enabled": true
```

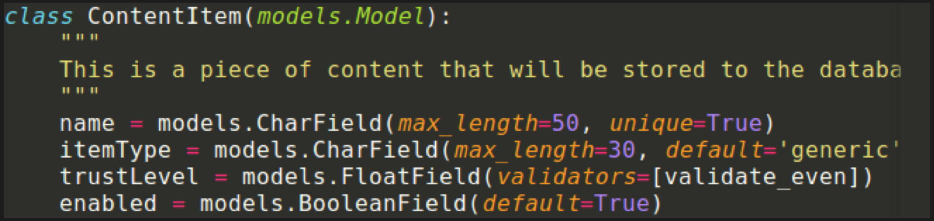
Django REST Framework

- Serializers
- Views / class-based views / viewsets
- router, simple urls
- multiple methods GET/POST/PUT/DELETE
- auto-documenting browseable API in markdown
- clear separation of code

Django REST Framework



```
class ContentItemSerializer(serializers.HyperlinkedModelSerializ
    class Meta:
        model = ContentItem
        fields = ('id','name', 'itemType', 'trustLevel', 'enable
```



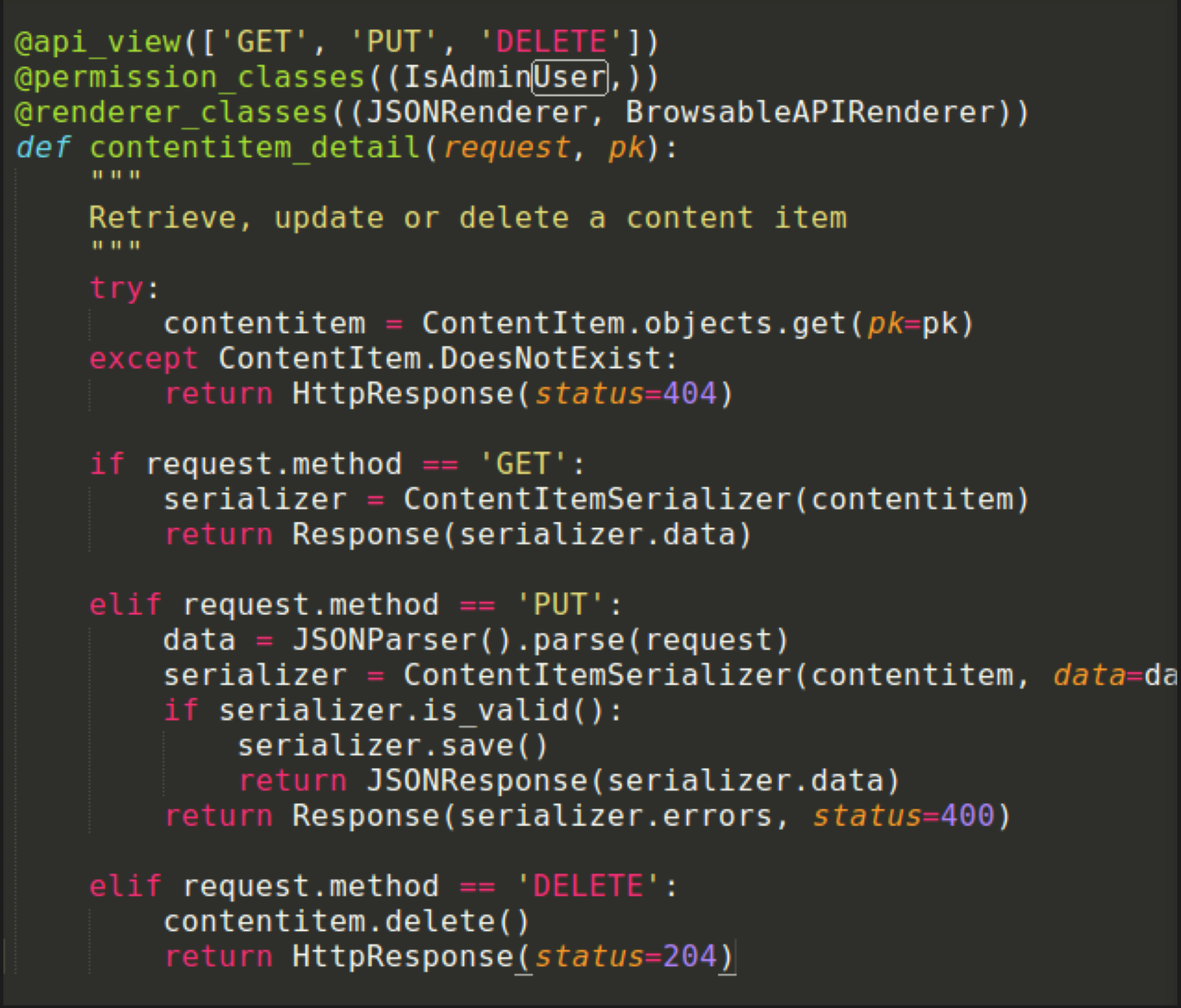
```
class ContentItem(models.Model):
    """
    This is a piece of content that will be stored to the databa
    """
    name = models.CharField(max_length=50, unique=True)
    itemType = models.CharField(max_length=30, default='generic'
    trustLevel = models.FloatField(validators=[validate_even])
    enabled = models.BooleanField(default=True)
```

Serializer

- map to a model or data type
- automagically serialize python data to JSON
- specify what fields to use and any more advanced features
- can use pre-built components or write your own

More info: http://www.django-rest-framework.org/api-guide/serializers

Django REST Framework

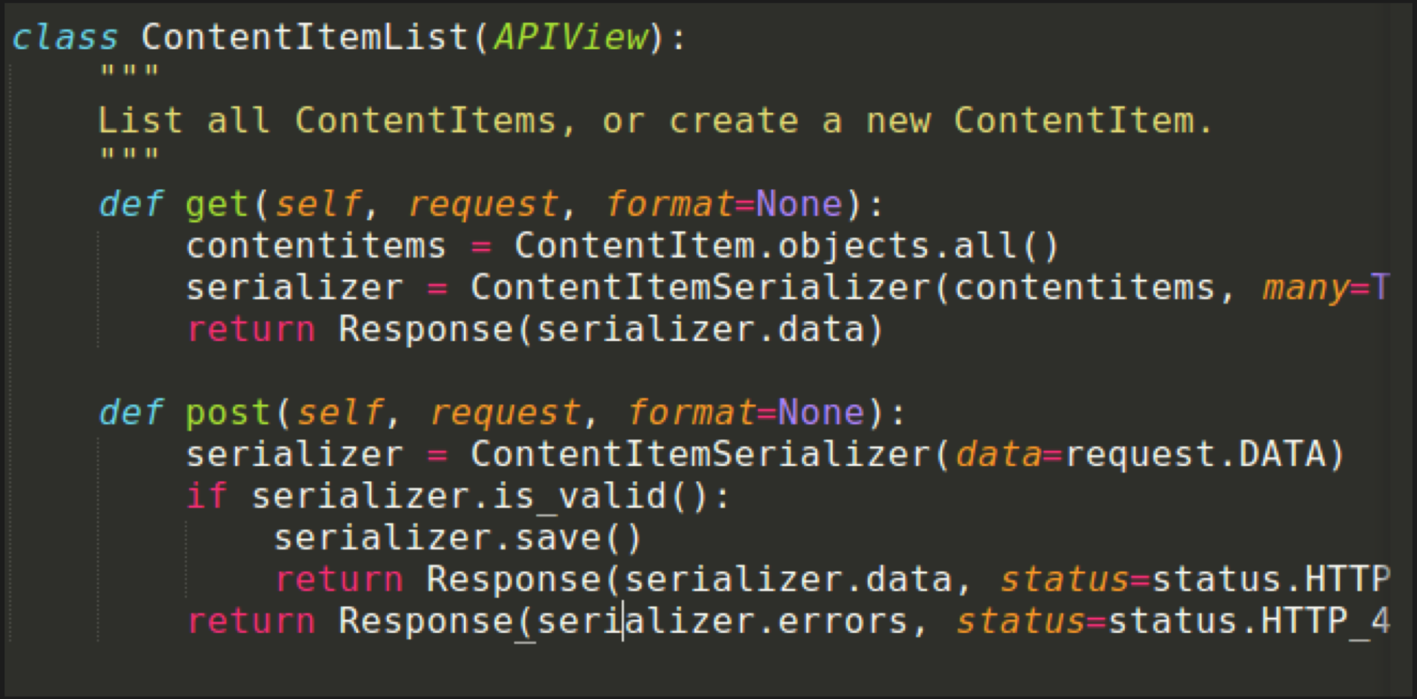


```
@api_view(['GET', 'PUT', 'DELETE'])
@permission_classes((IsAdminUser,))
@renderer_classes((JSONRenderer, BrowsableAPIRenderer))
def contentitem_detail(request, pk):
    """
    Retrieve, update or delete a content item
    """
    try:
        contentitem = ContentItem.objects.get(pk=pk)
    except ContentItem.DoesNotExist:
        return HttpResponse(status=404)

    if request.method == 'GET':
        serializer = ContentItemSerializer(contentitem)
        return Response(serializer.data)

    elif request.method == 'PUT':
        data = JSONParser().parse(request)
        serializer = ContentItemSerializer(contentitem, data=da
        if serializer.is_valid():
            serializer.save()
            return JSONResponse(serializer.data)
        return Response(serializer.errors, status=400)

    elif request.method == 'DELETE':
        contentitem.delete()
        return HttpResponse(status=204)
```

Simple function-based views

- lowest level way to dictate an API call
- highest amount of code
- more prone to errors
- use only if you need to provide very specific functionality

Django REST Framework



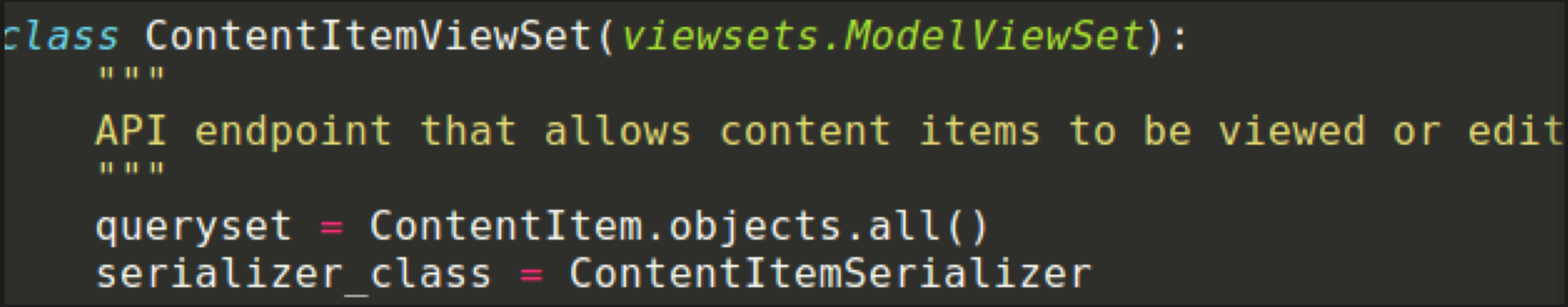
```
class ContentItemList(APIView):
    """
    List all ContentItems, or create a new ContentItem.
    """
    def get(self, request, format=None):
        contentitems = ContentItem.objects.all()
        serializer = ContentItemSerializer(contentitems, many=T
        return Response(serializer.data)

    def post(self, request, format=None):
        serializer = ContentItemSerializer(data=request.DATA)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP
        return Response(serializer.errors, status=status.HTTP_4
```

Class-based views

- higher level way to dictate an API call
- better way to group requests
- Still requires effort to create each handler

More info: http://www.django-rest-framework.org/api-guide/views

Django REST Framework



```
class ContentItemViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows content items to be viewed or edit
    """
    queryset = ContentItem.objects.all()
    serializer_class = ContentItemSerializer
```

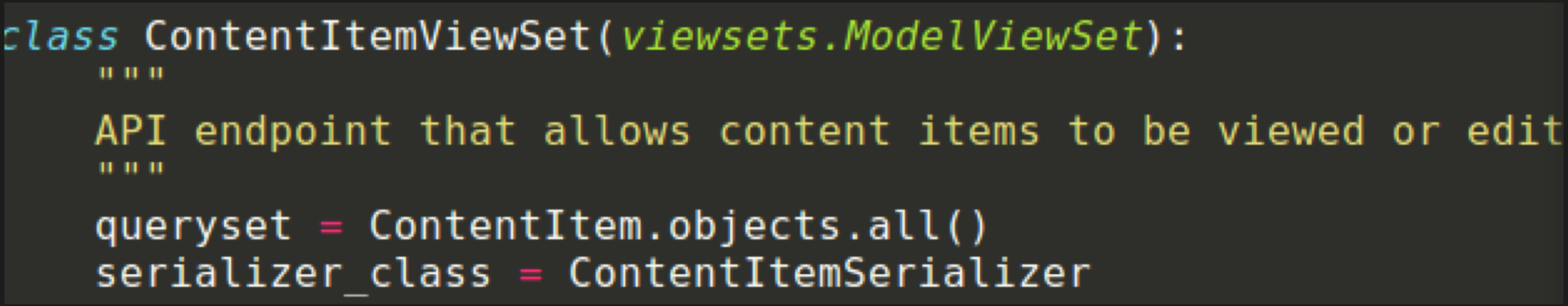
Viewsets

- very high level way of dictating API calls
- DRF Automagically generates multiple views that map to GET,POST, etc
- can still be overridden
- This is the “quick and easy” way to get an API up, but you have less control

Django REST Framework

More on Viewsets

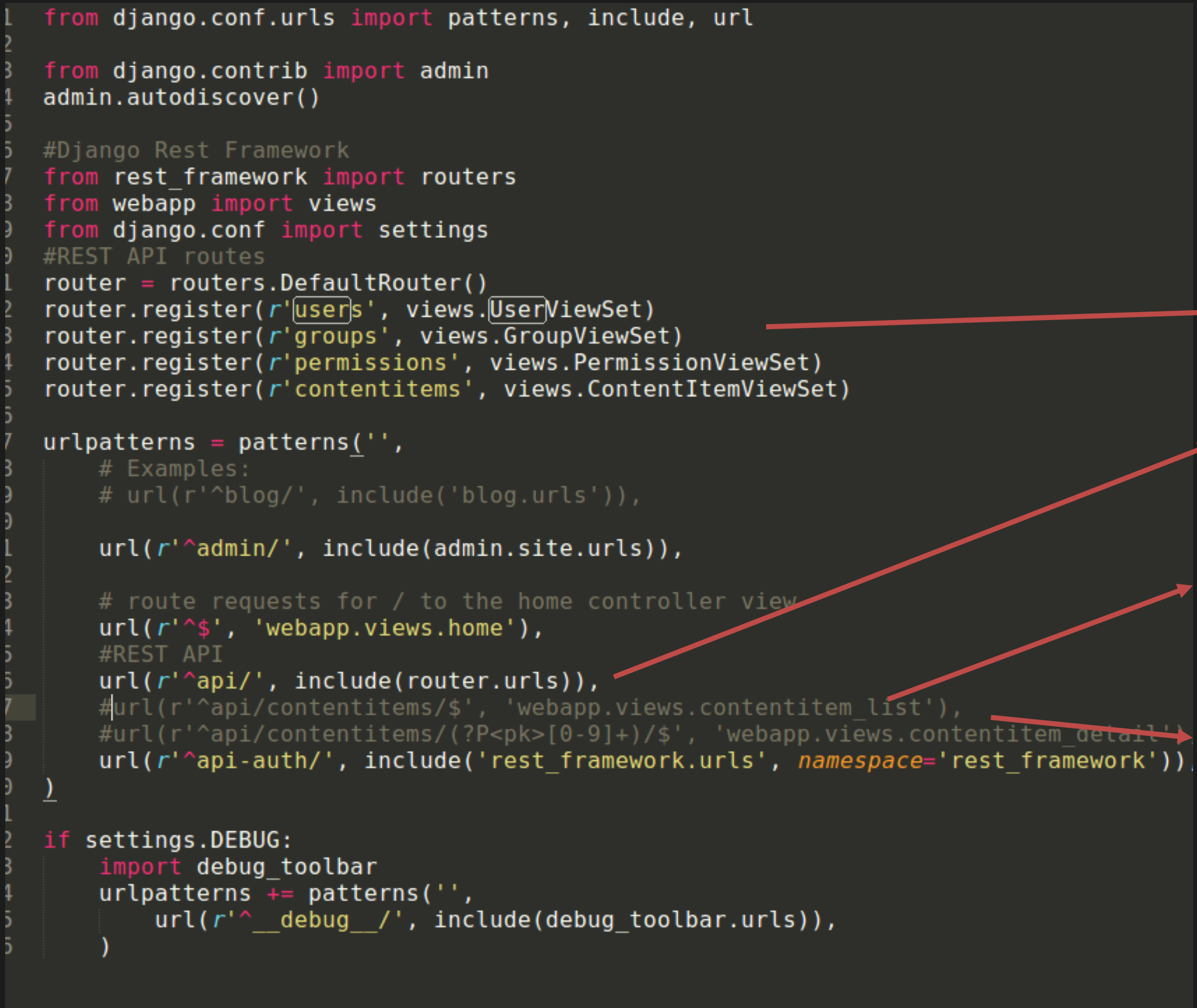
- queryset map to a set of database models
- creates views to handle GET/POST/ETC requests to /contentitems/ and /contentitems/<pk>
- serializer_class parses the data for the related views
- can specify new methods as function e.g. def foo on in a viewset to handle special cases or perform functions like /contentitems/<pk>/foo
- can override base views using list, create, retrieve, update, partial_update, and destroy keywords these map to HTTP methods



```
class ContentItemViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows content items to be viewed or edit
    """
    queryset = ContentItem.objects.all()
    serializer_class = ContentItemSerializer
```

More info: http://www.django-rest-framework.org/api-guide/viewsets

Django REST Framework



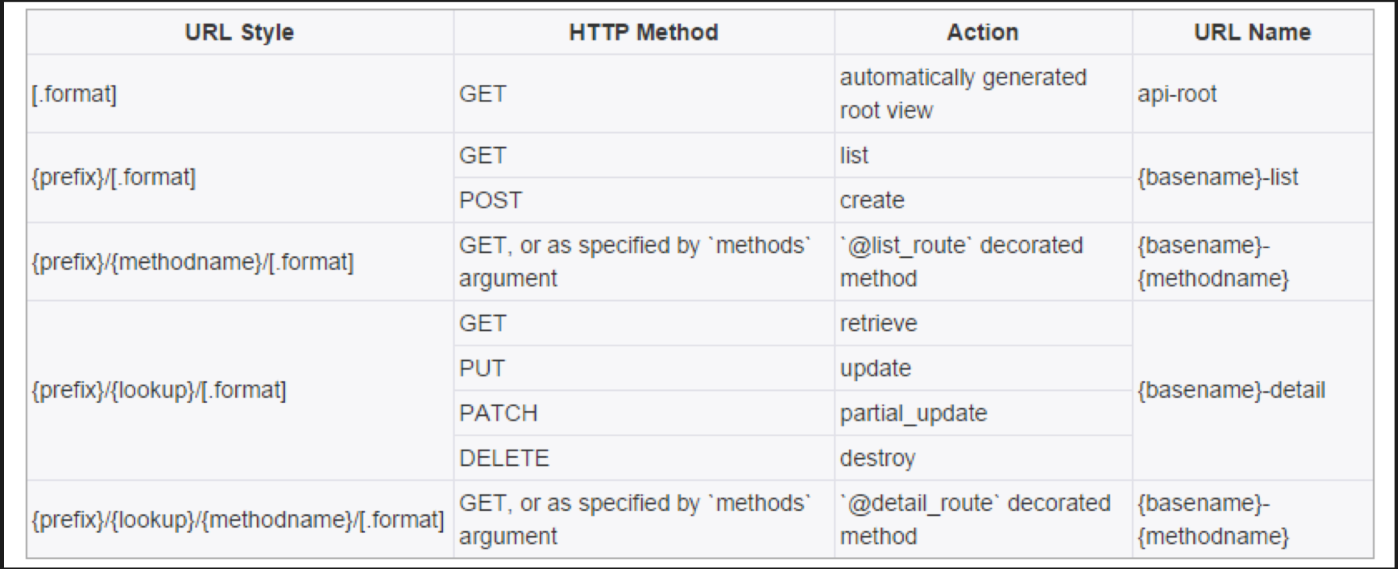
```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

#Django Rest Framework
from rest_framework import routers
from webapp import views
from django.conf import settings
#REST API routes
router = routers.DefaultRouter()
router.register(r'users', views.UserViewSet)
router.register(r'groups', views.GroupViewSet)
router.register(r'permissions', views.PermissionViewSet)
router.register(r'contentitems', views.ContentItemViewSet)

urlpatterns = patterns('',
    # Examples:
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),

    # route requests for / to the home controller view
    url(r'^$', 'webapp.views.home'),
    #REST API
    url(r'^api/', include(router.urls)),
    #url(r'^api/contentitems/$', 'webapp.views.contentitem_list'),
    #url(r'^api/contentitems/(?P<pk>[0-9]+)/$', 'webapp.views.contentitem_detail'),
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework')),
)

if settings.DEBUG:
    import debug_toolbar
    urlpatterns += patterns('',
        url(r'^__debug__/', include(debug_toolbar.urls)),
    )
```

Wiring the API with URLs

- Viewsets
  - Can be customized
- Use router for connecting viewsets to urls
- Can use view mapping for class-based views
- Can use basic URLs for function-based views

Wiring the API with URLs: Using the Router

- prefix is specified in the .register call.
- E.g. `router.register(r'contentitems', views.ContentItemViewSet)`
- methodname is a custom method detailed in the viewset
- lookup is the primary key or other unique field that identifies one instance



URL Style	HTTP Method	Action	URL Name
[.format]	GET	automatically generated root view	api-root
{prefix}/[.format]	GET	list	{basename}-list
	POST	create	
{prefix}/{methodname}/[.format]	GET, or as specified by `methods` argument	`@list_route` decorated method	{basename}-{methodname}
{prefix}/{lookup}/[.format]	GET	retrieve	{basename}-detail
	PUT	update	
	PATCH	partial_update	
	DELETE	destroy	
{prefix}/{lookup}/{methodname}/[.format]	GET, or as specified by `methods` argument	`@detail_route` decorated method	{basename}-{methodname}

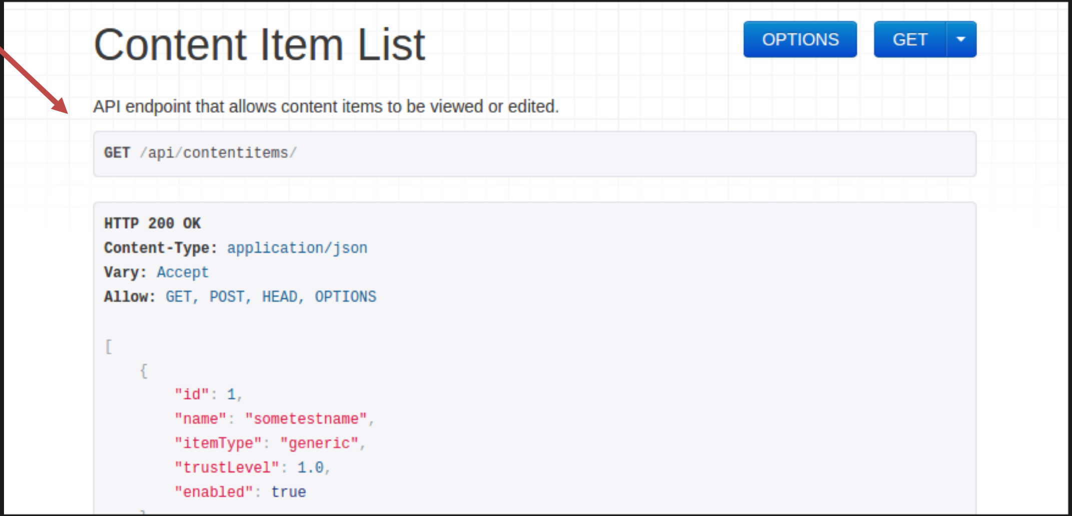
More info: http://www.django-rest-framework.org/api-guide/routers

D
j F
a r
n a
g m
o e
w
R o
E r
S k
T

Auto-magical Documentation

- Whatever pydocs comments you make are translated using markdown into HTML automagically

```
class ContentItemViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows content items to be viewed or edited.
    """
    queryset = ContentItem.objects.all()
    serializer_class = ContentItemSerializer
```



Content Item List

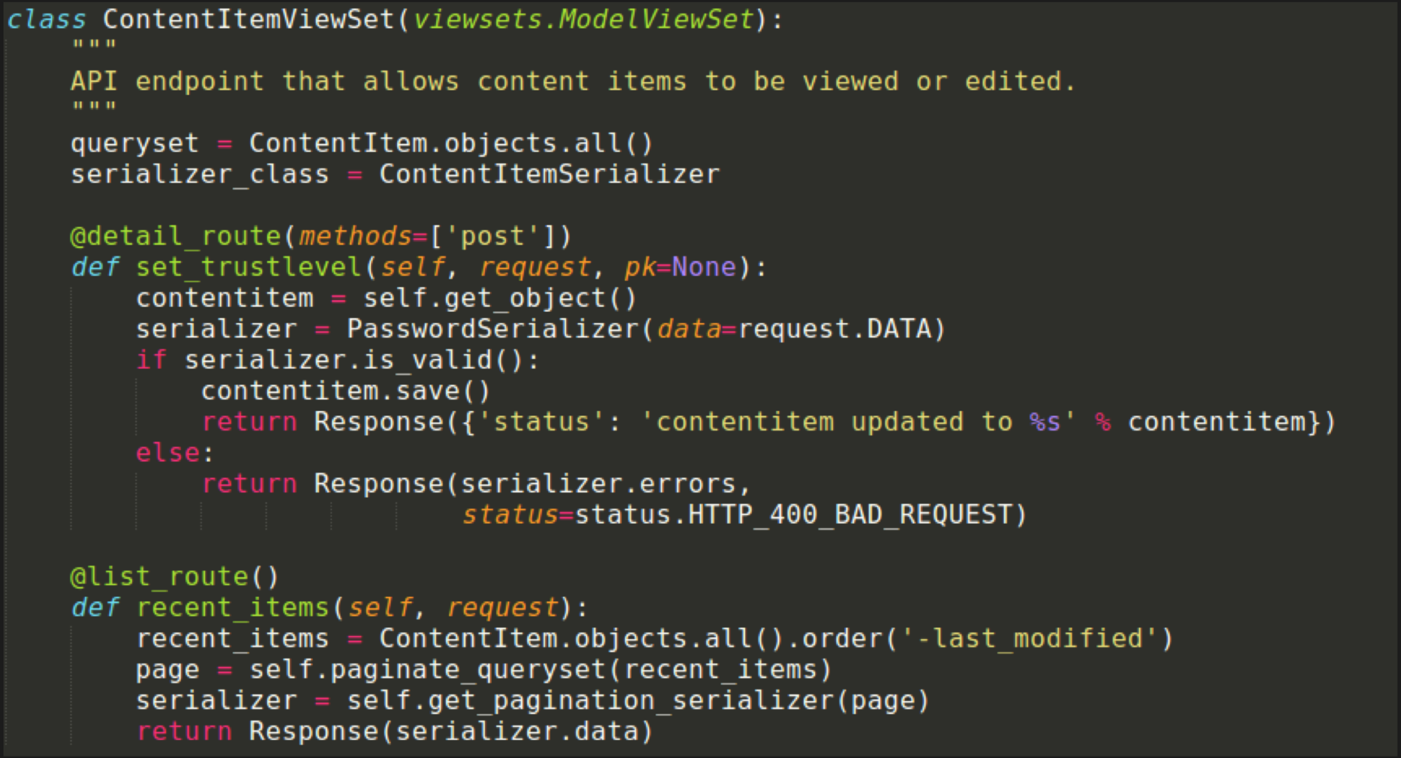
API endpoint that allows content items to be viewed or edited.

```
GET /api/contentitems/
```

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

[
    {
        "id": 1,
        "name": "sometestname",
        "itemType": "generic",
        "trustLevel": 1.0,
        "enabled": true
```

Self Documenting Browsable API

- use detail_route() for individual items
- use list_route() for all items



```python
class ContentItemViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows content items to be viewed or edited.
    """
    queryset = ContentItem.objects.all()
    serializer_class = ContentItemSerializer

    @detail_route(methods=['post'])
    def set_trustlevel(self, request, pk=None):
        contentitem = self.get_object()
        serializer = PasswordSerializer(data=request.DATA)
        if serializer.is_valid():
            contentitem.save()
            return Response({'status': 'contentitem updated to %s' % contentitem})
        else:
            return Response(serializer.errors,
                            status=status.HTTP_400_BAD_REQUEST)

    @list_route()
    def recent_items(self, request):
        recent_items = ContentItem.objects.all().order('-last_modified')
        page = self.paginate_queryset(recent_items)
        serializer = self.get_pagination_serializer(page)
        return Response(serializer.data)
```

Questions?

Matt Hale, PhD

University of Nebraska at Omaha

Interdisciplinary Informatics

mlhale@unomaha.edu

Twitter: @mlhale_