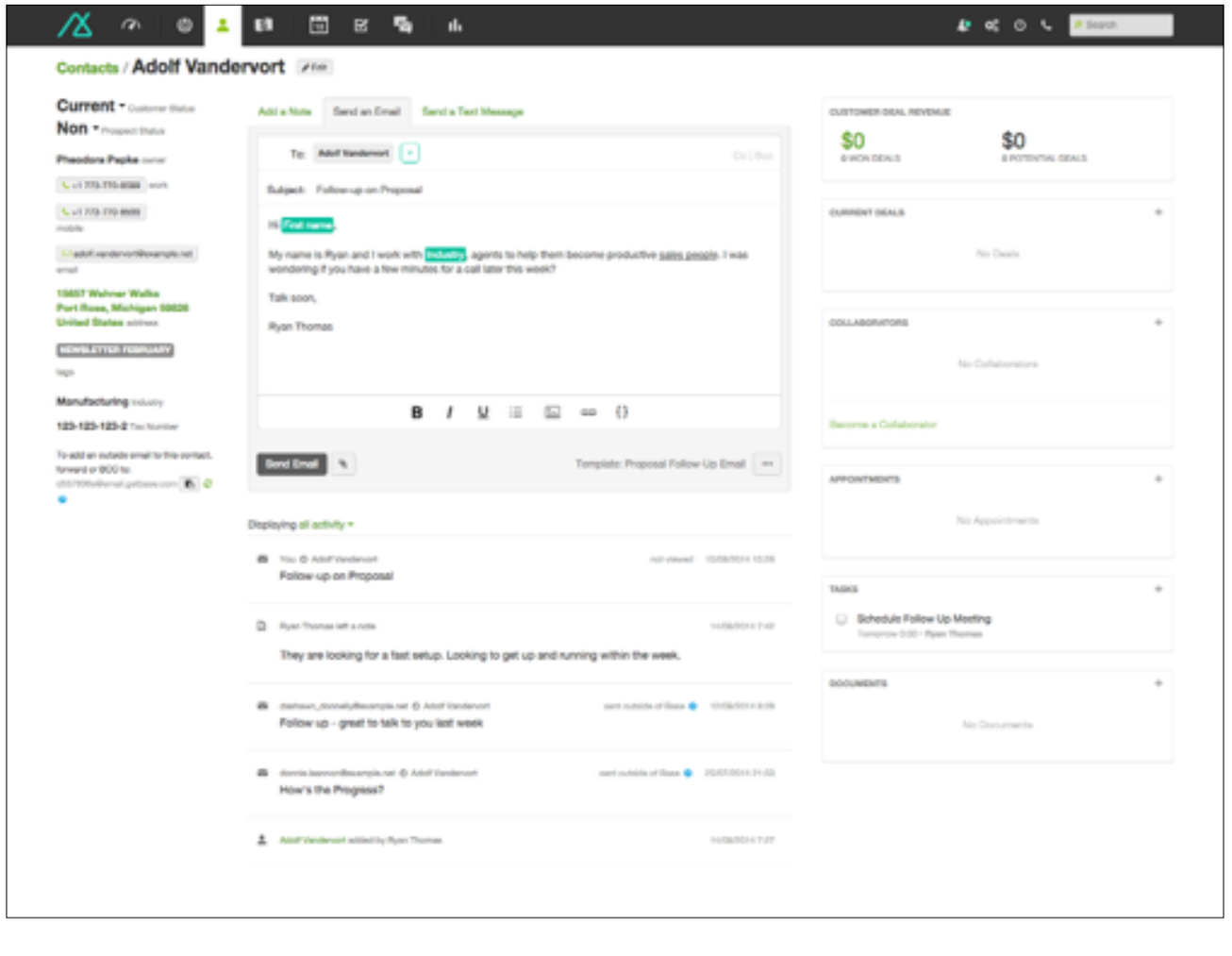
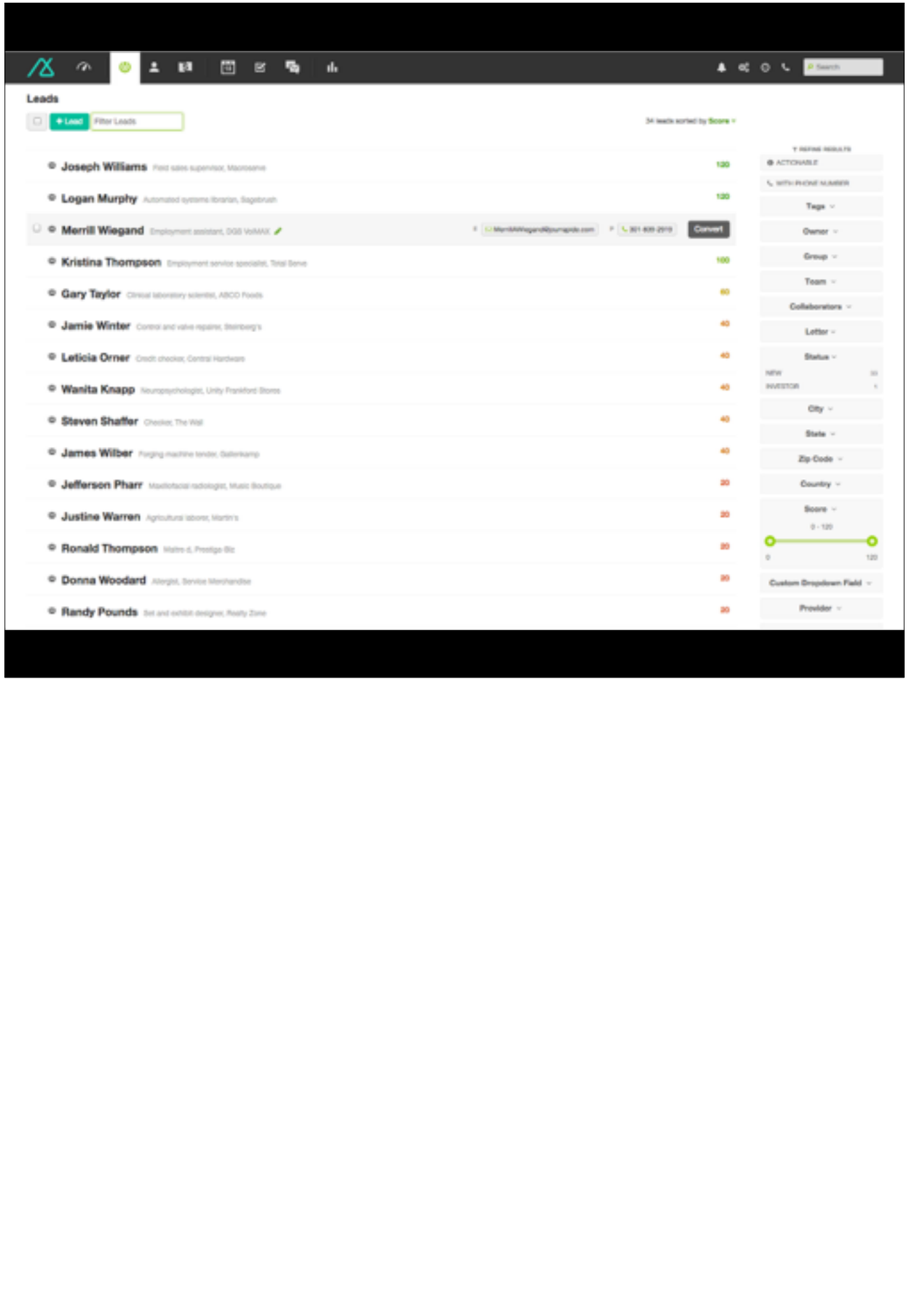
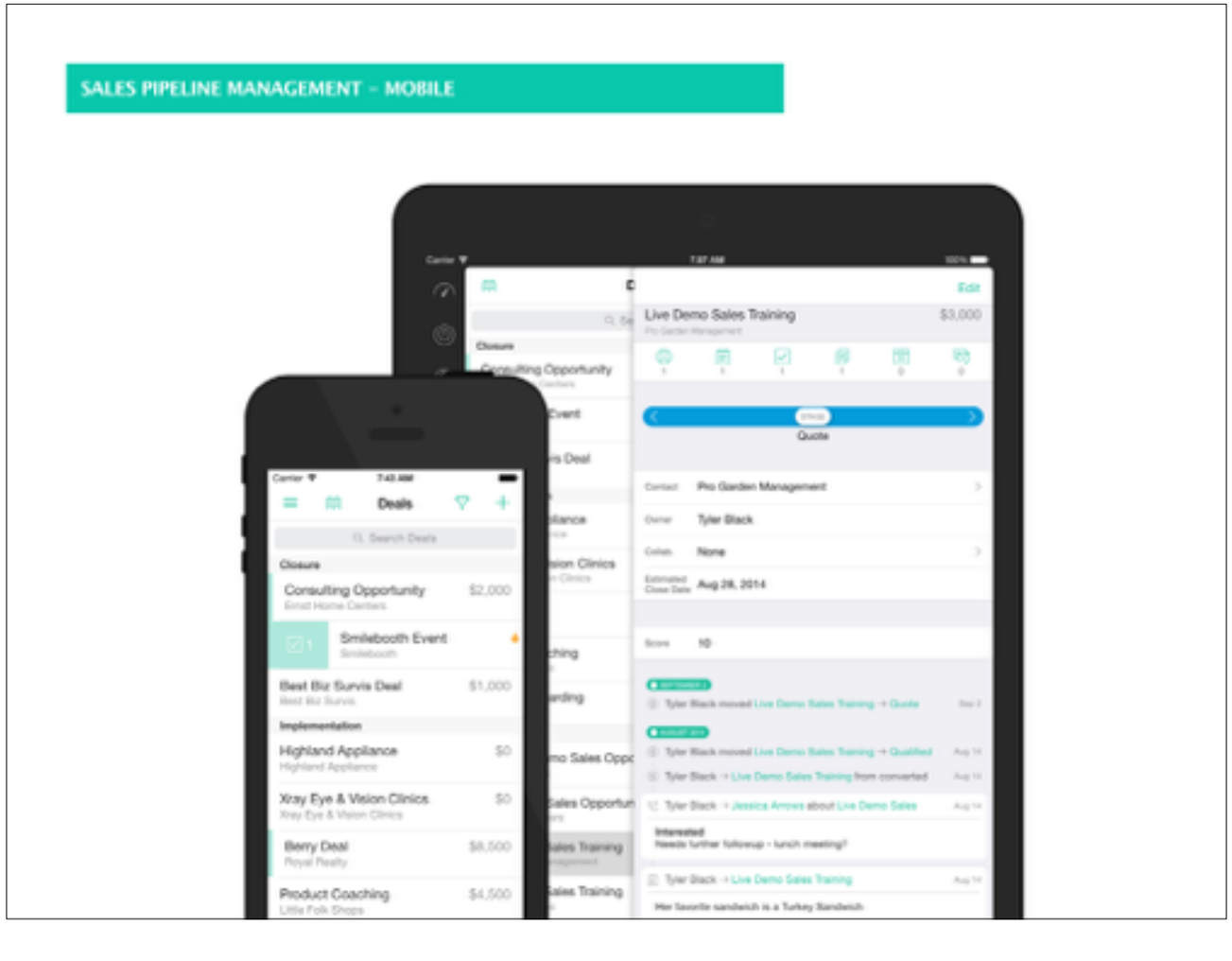
Hi everyone! Or should I say, cześć! My name is Rodolfo, I'm a developer at Base and today I'm gonna show you how to leverage the power of DRF to build Amazing Web APIs in Python.



Before we dive into the details, let me give you some context. Base is an Intelligent Sales Productivity Platform, it's a product built to make sales people 10x more productive. Here's how it looks like.

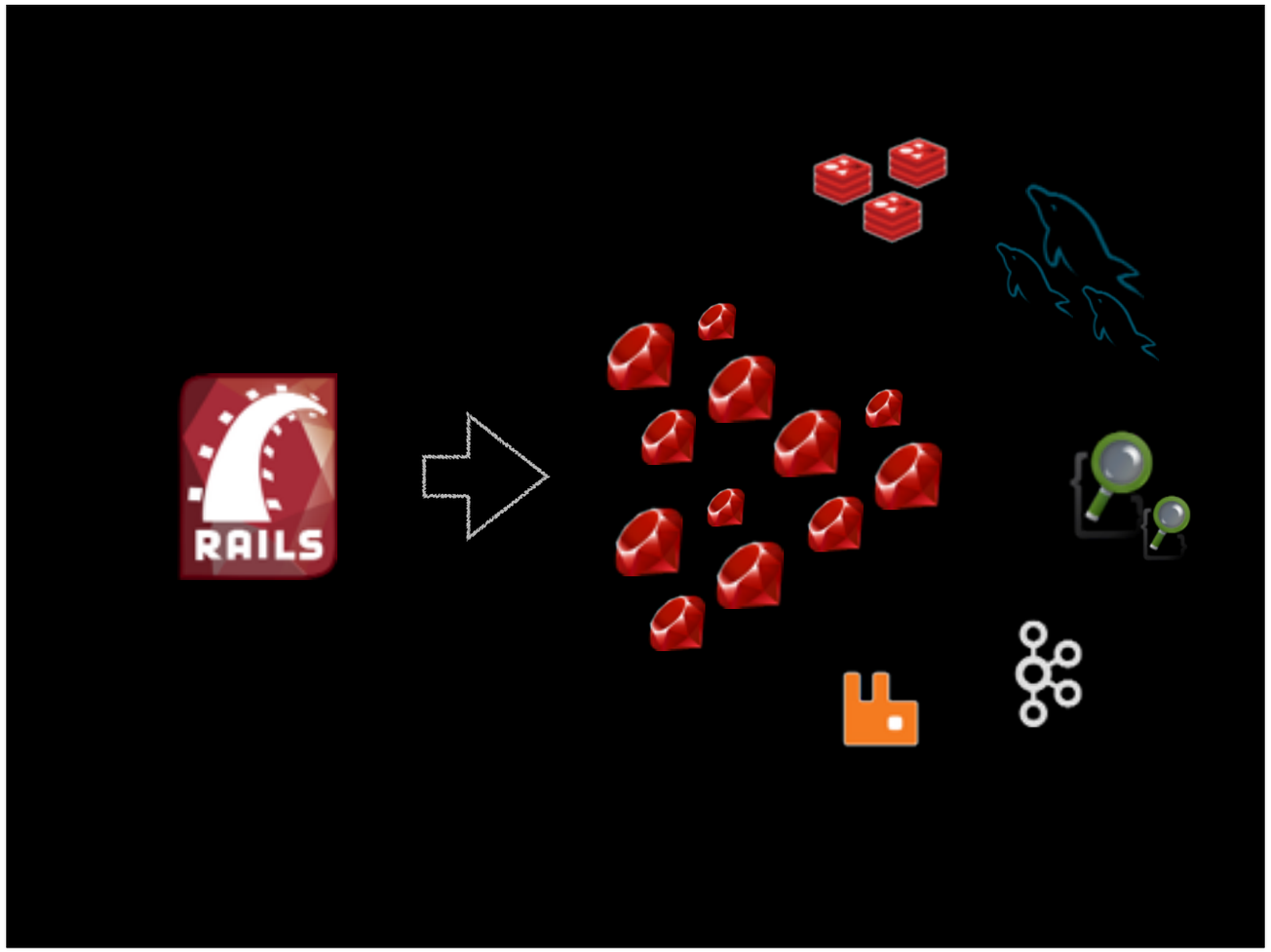

Leads
+ Lead
Filter Leads
34 leads sorted by Score
Joseph Williams
Logan Murphy
Merrill Wiegand
Convert
Kristina Thompson
Gary Taylor
Jamie Winter
Leticia Orner
Wanita Knapp
Steven Shaffer
James Wilber
Jefferson Pharr
Justine Warren
Ronald Thompson
Donna Woodard
Randy Pounds
Tags
Owner
Group
Team
Collaborators
Letter
Status
City
State
Zip Code
Country
Score
Custom Dropdown Field
Provider



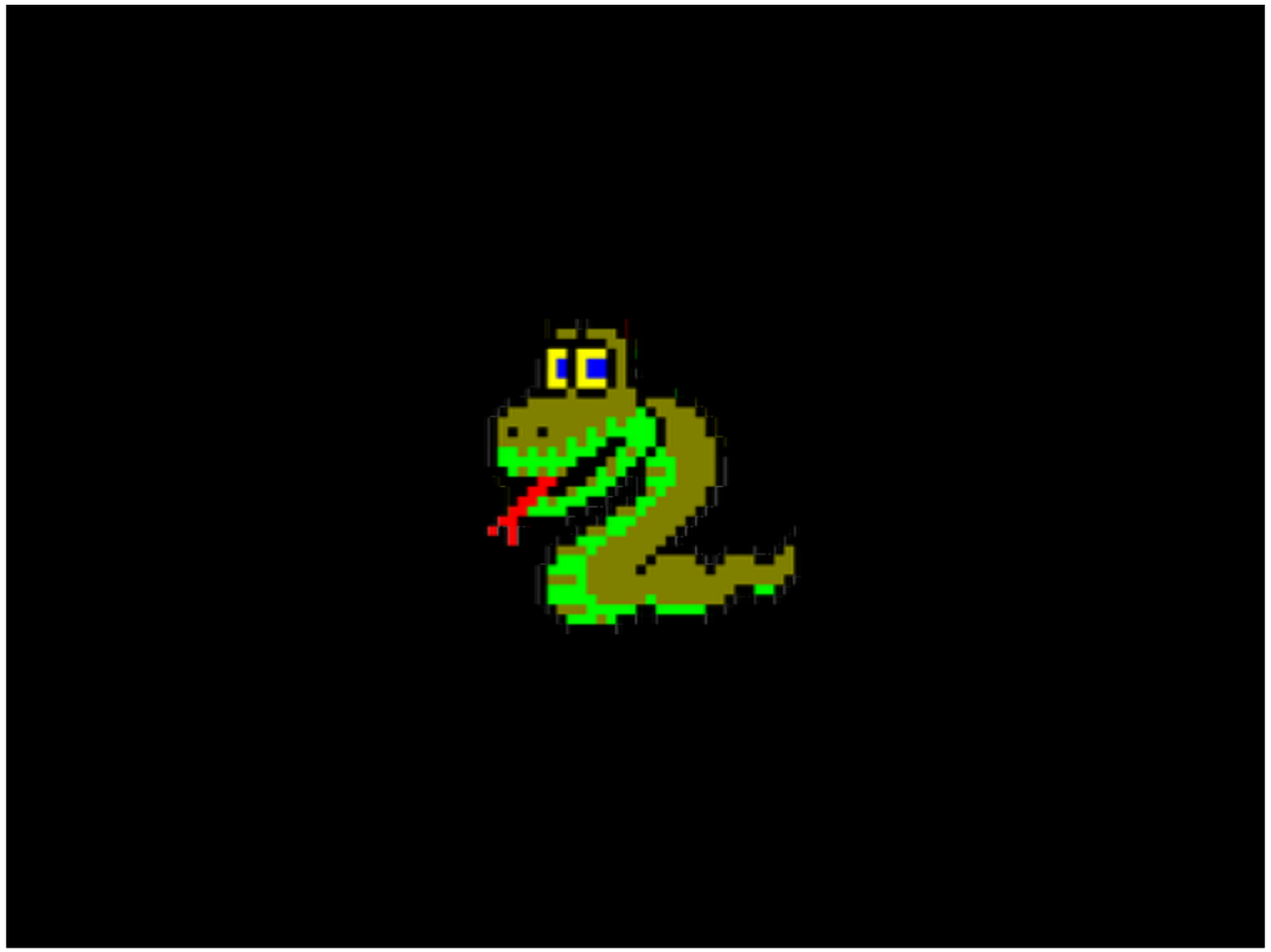
Base is available on the Web and all majors mobile platforms.

In the early days, we were powered by a monolithic stack based on Ruby on Rails. It was a good way to get started really quick and position our product. Our culture emphasizes that “it’s null till you ship it”.

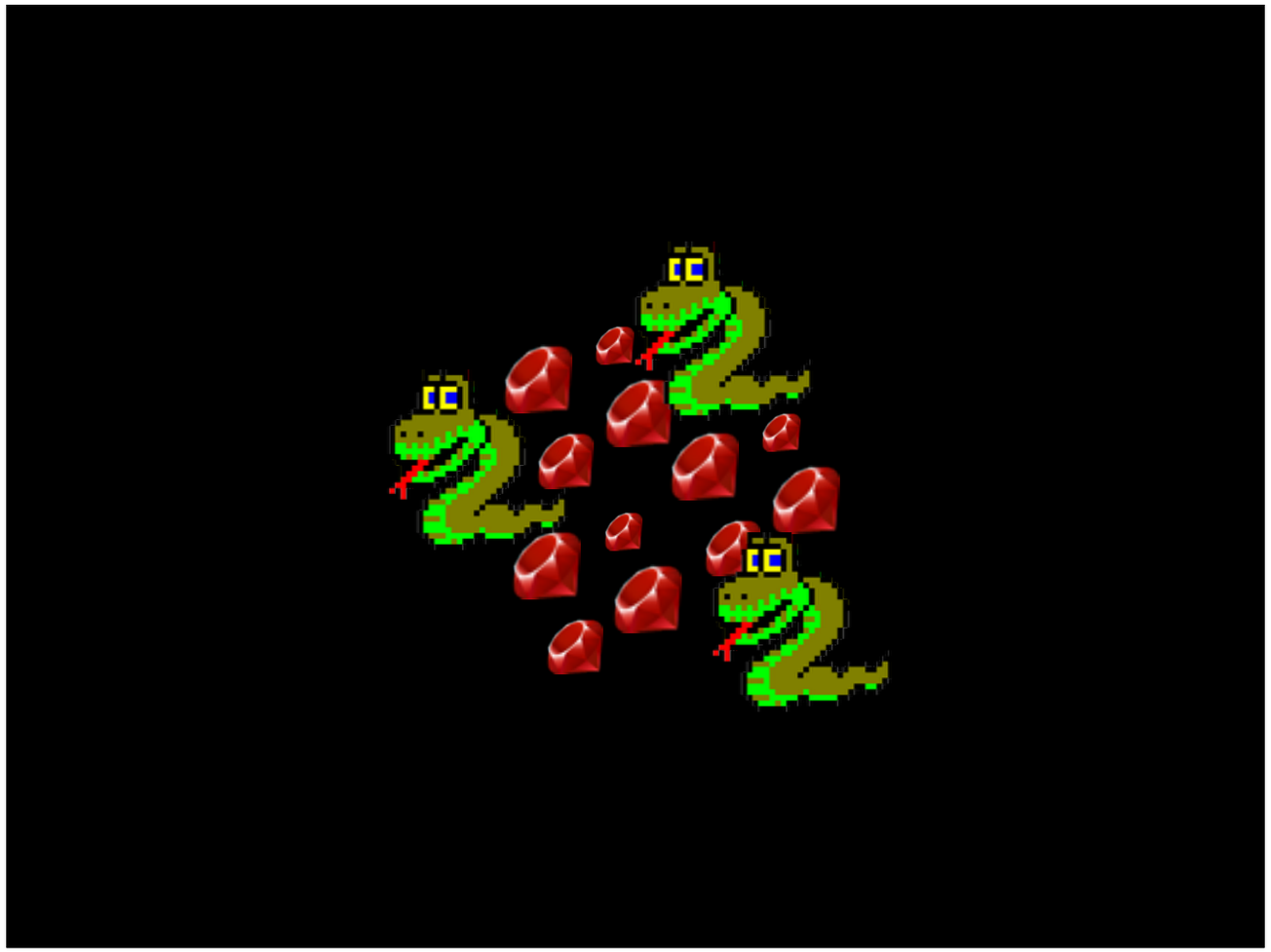
But then… we happily achieved the point where the monolithic approach won’t allow us to go forward.



That’s when we decomposed our system into many parts, forming a distributed system of micro services that implement Base and deliver value to our customers. This system is powered by many supporting & enabling pieces like database clusters and message queues, and tens of web services, initially written in Ruby.



In the infrastructure we had, there was nothing in particular that forced us to use Ruby to implement those services. In the process of growing the company we introduced Python last November (almost a year now).



Now our system is more like this (we also have parts in other languages!). We have the challenge to cope with the frenetic pace of delivering features in our product.

In services written in Python, DRF is a key technology that enables us to move fast. And I'm gonna show how it will empower you to do the same.

The title of this talk is “Amazing Web APIs”. But what the heck is an Amazing Web API? For the context of this talk, I’m gonna focus on ‘pragmatic JSON APIs’, and leave aside the purity of REST, hyperlinked data and a lot of other cool things. If you wanna talk about those, find me at the corridor or at the Base stand. Amazing because it is easy!

service.getbase.com

```
GET     /api/v1/deals
GET     /api/v1/deals/42
POST    /api/v1/deals
PUT     /api/v1/deals/42
DELETE  /api/v1/deals/42
```

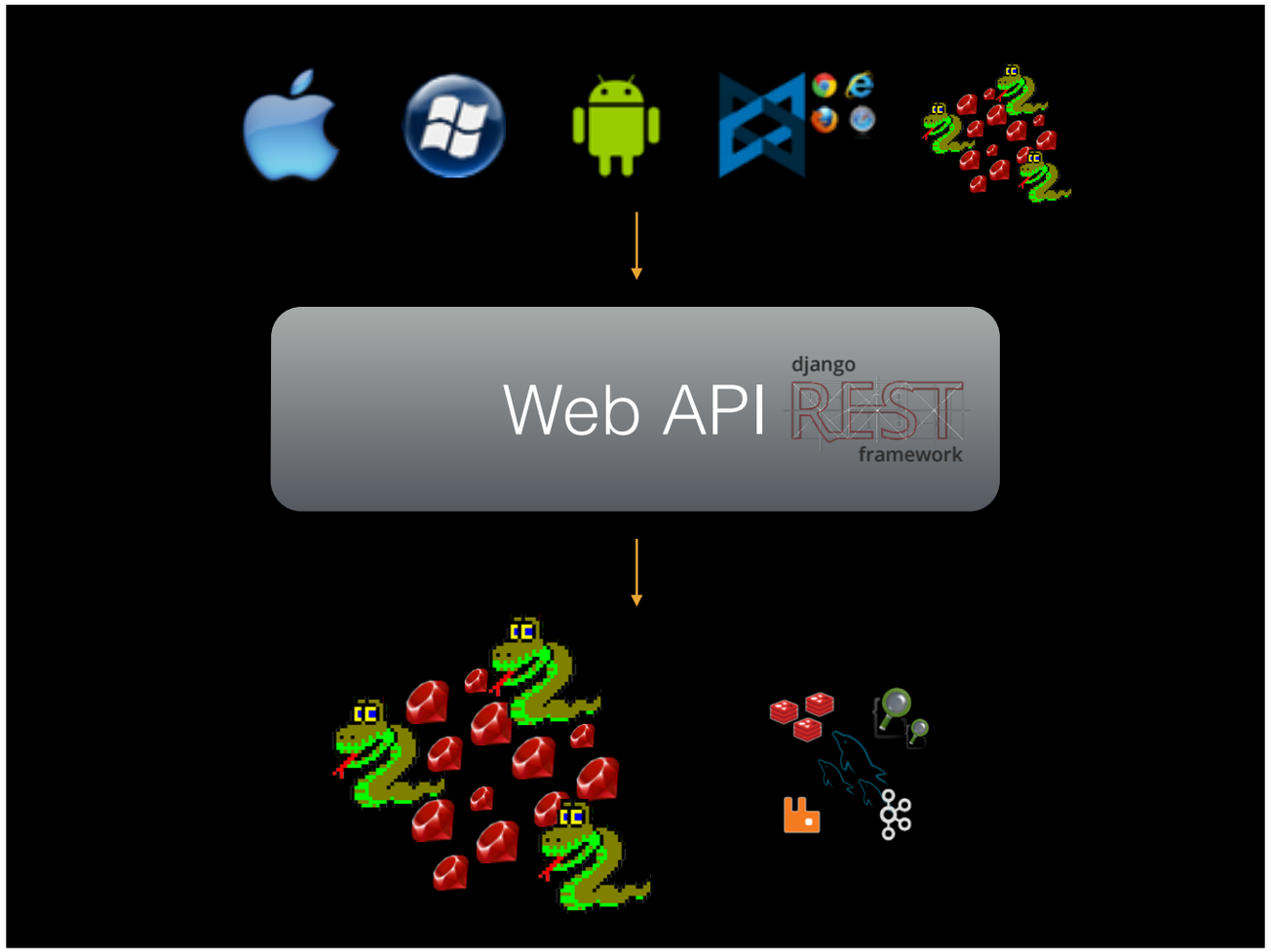
The pragmatic APIs we're focused on are those in which the input is an HTTP request (with an optional JSON payload) to some endpoint in our Django application and the response is a piece of JSON like this one. A client can either consume JSON to show some UI, or send JSON to update state in the server.

service.getbase.com

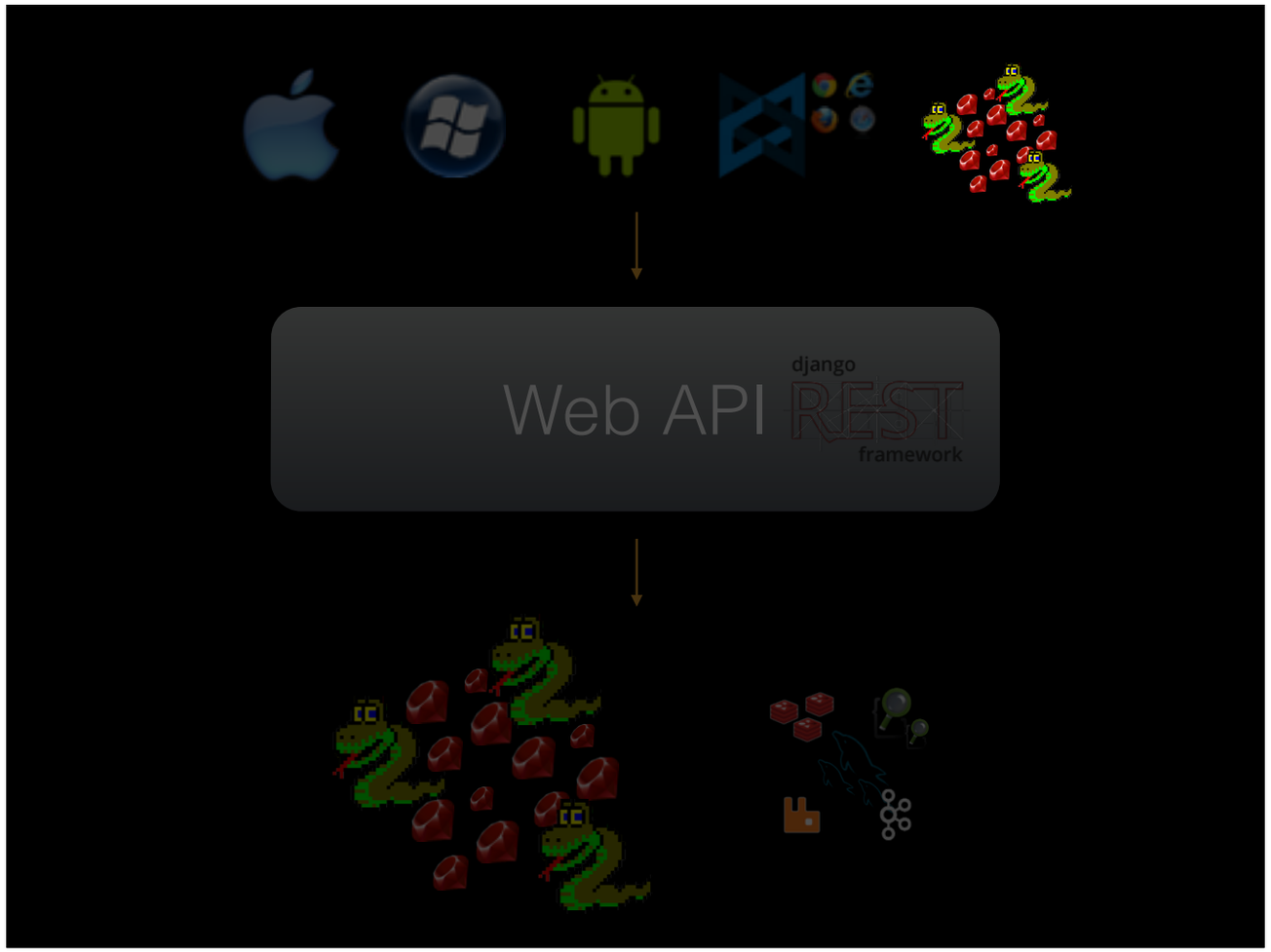
```
GET     /api/v1/deals
GET     /api/v1/deals/42
POST    /api/v1/deals
PUT     /api/v1/deals/42
DELETE  /api/v1/deals/42
```


```
{
  "deal": {
    "id": 1,
    "name": "Ryan T. Smith",
    "value": 42,
    "updated_at": "2014-10-13T14:37:10Z"
  }
}
```

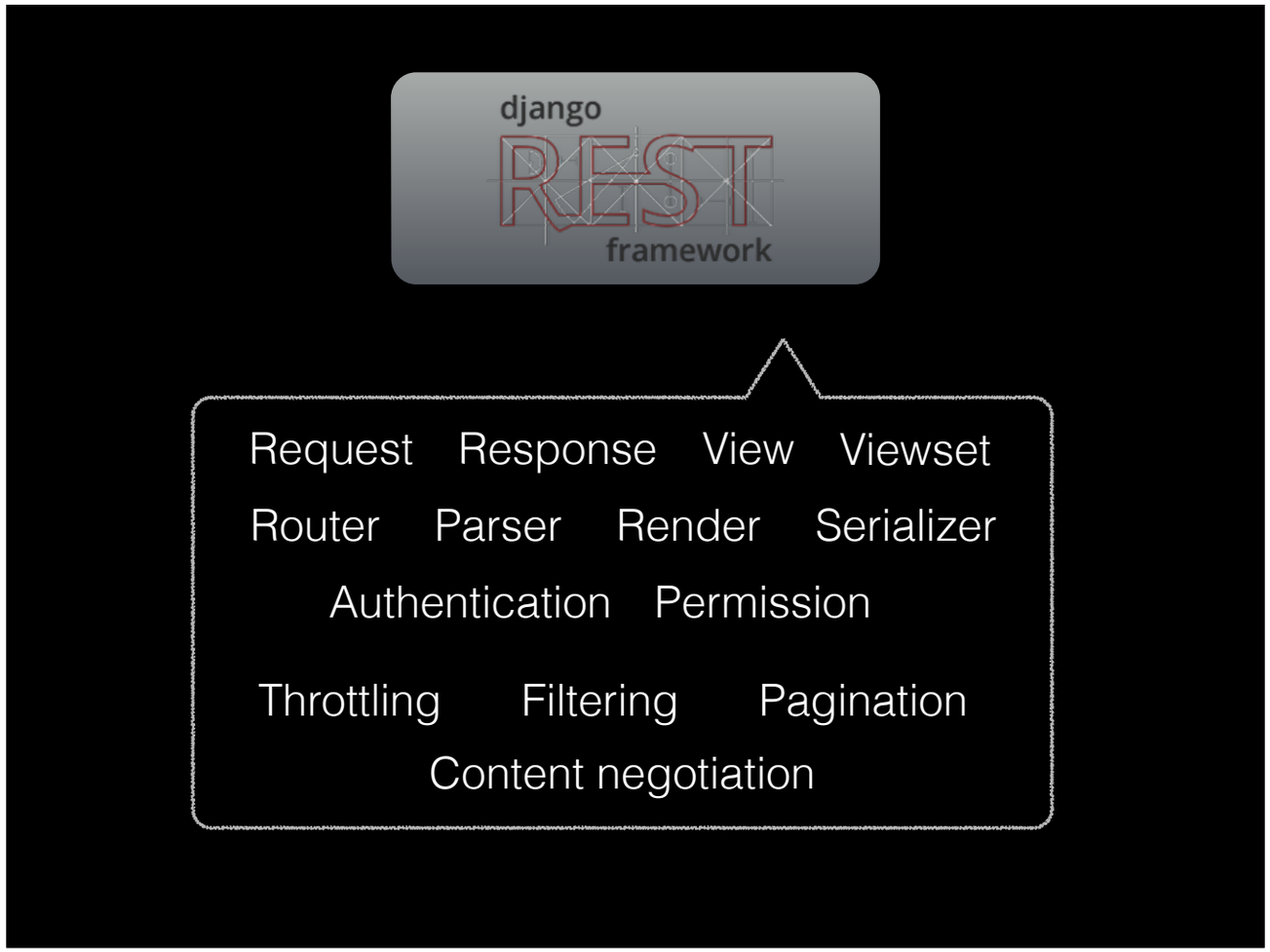

The pragmatic APIs we're focused on are those in which the input is an HTTP request (with an optional JSON payload) to some endpoint in our Django application and the response is a piece of JSON like this one. A client can either consume JSON to show some UI, or send JSON to update state in the server.



The Web API is therefore the layer that allows us to serve multiple types of clients consistently and scale the development of our product. Scaling in terms of performance is important, but for us it's also fundamental to be able to ship fast. Notice how we eat our own dog food, services interact with other services via the very same set of APIs.



The Web API is therefore the layer that allows us to serve multiple types of clients consistently and scale the development of our product. Scaling in terms of performance is important, but for us it's also fundamental to be able to ship fast. Notice how we eat our own dog food, services interact with other services via the very same set of APIs.



DRF brings in a bunch of concepts, we'll drive slowly through each part explaining how it all fits together.

For a hands-on experience come code with us at the Base stand.

```
$ django-admin.py startproject adrf
$ ./manage.py startapp api
$ tree
.
├── adrf
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── api
│   ├── __init__.py
│   ├── admin.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── requirements.txt
```

We start from a regular django project and app. I called my project “adrf” and the app “api”.

```
$ cat requirements.txt
Django==1.7
djangorestframework==2.4.3
```

We start from a regular django project and app. I called my project “adrf” and the app “api”.

api/models.py

```
class Deal(models.Model):
    name = models.CharField(max_length=255)
    value = models.IntegerField()
    updated_at = models.DateTimeField(auto_now=True)
```

api/serializers.py

```
class DealSerializer(serializers.ModelSerializer):
    class Meta:
        model = Deal
```

api/views.py

```
class DealViewSet(viewsets.ModelViewSet):
    queryset = Deal.objects.all()
    serializer_class = DealSerializer
```

api/urls.py

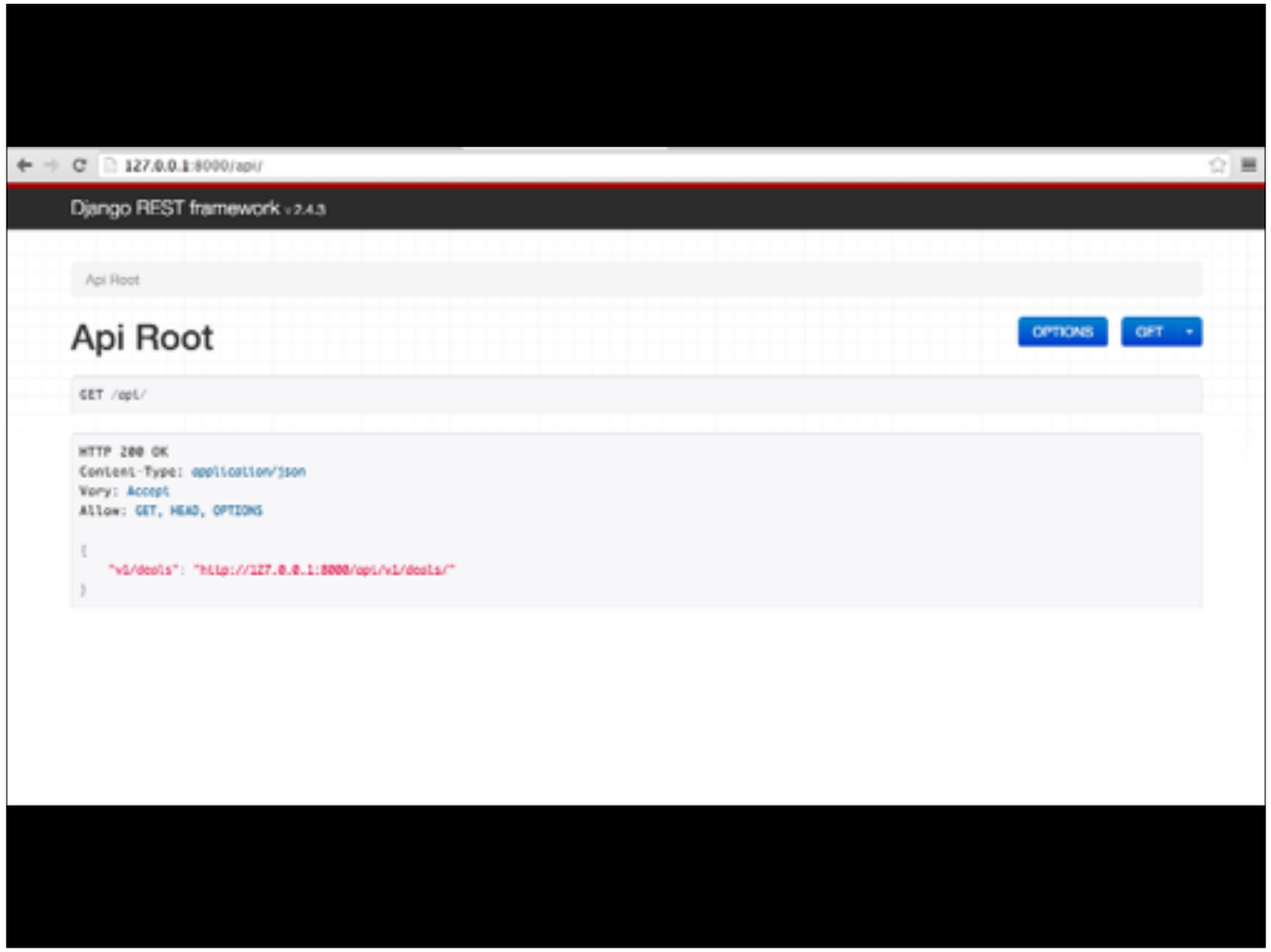
```
from rest_framework import routers
from api import views


router = routers.DefaultRouter()
router.register(r'v1/deals', views.DealsViewSet)

urlpatterns = router.urls
```

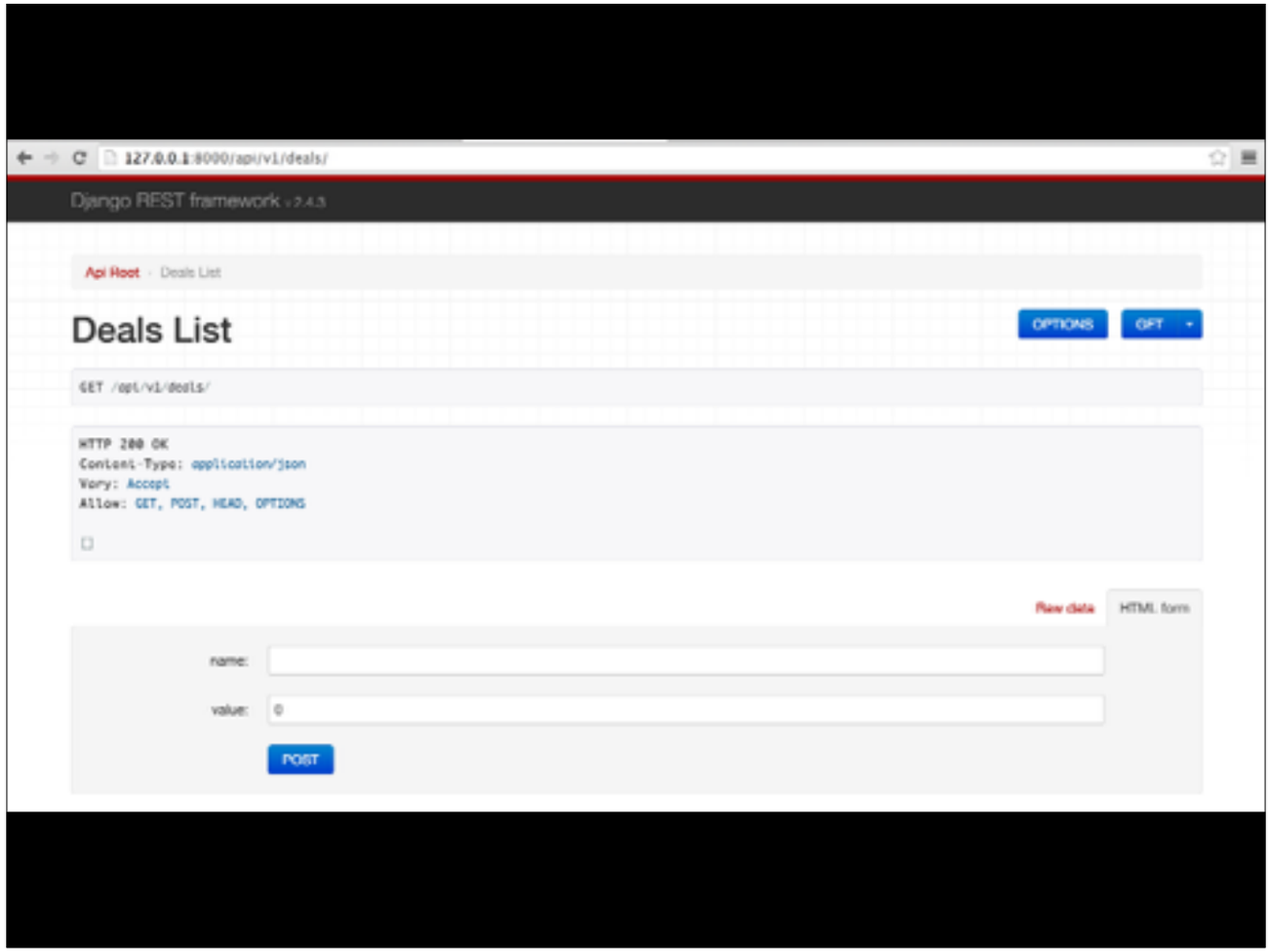
adrf/urls.py

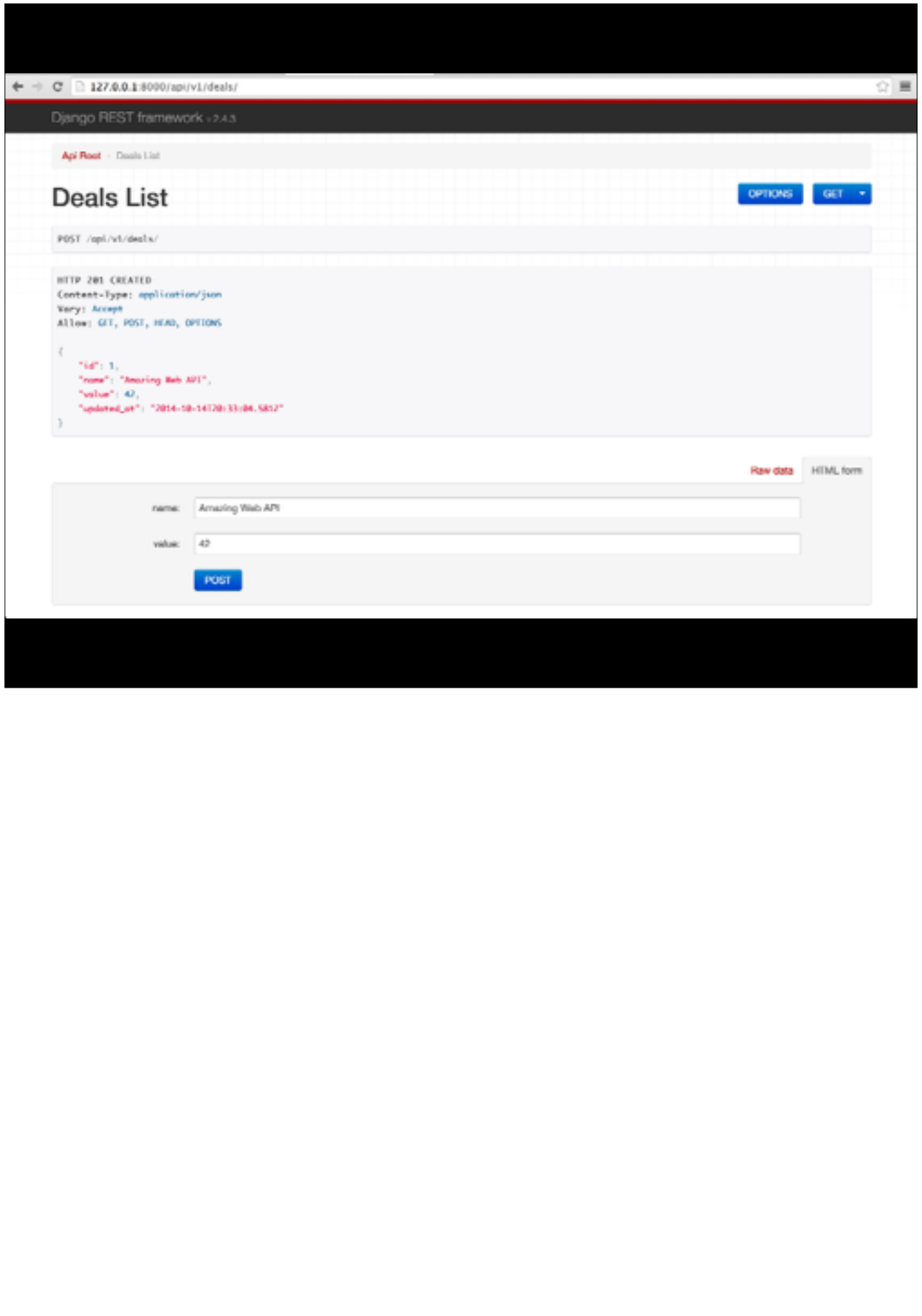
```
# ...

urlpatterns = patterns('',
    url(r'^api/', include('api.urls')),
    # ...
)
```

adrf/settings.py

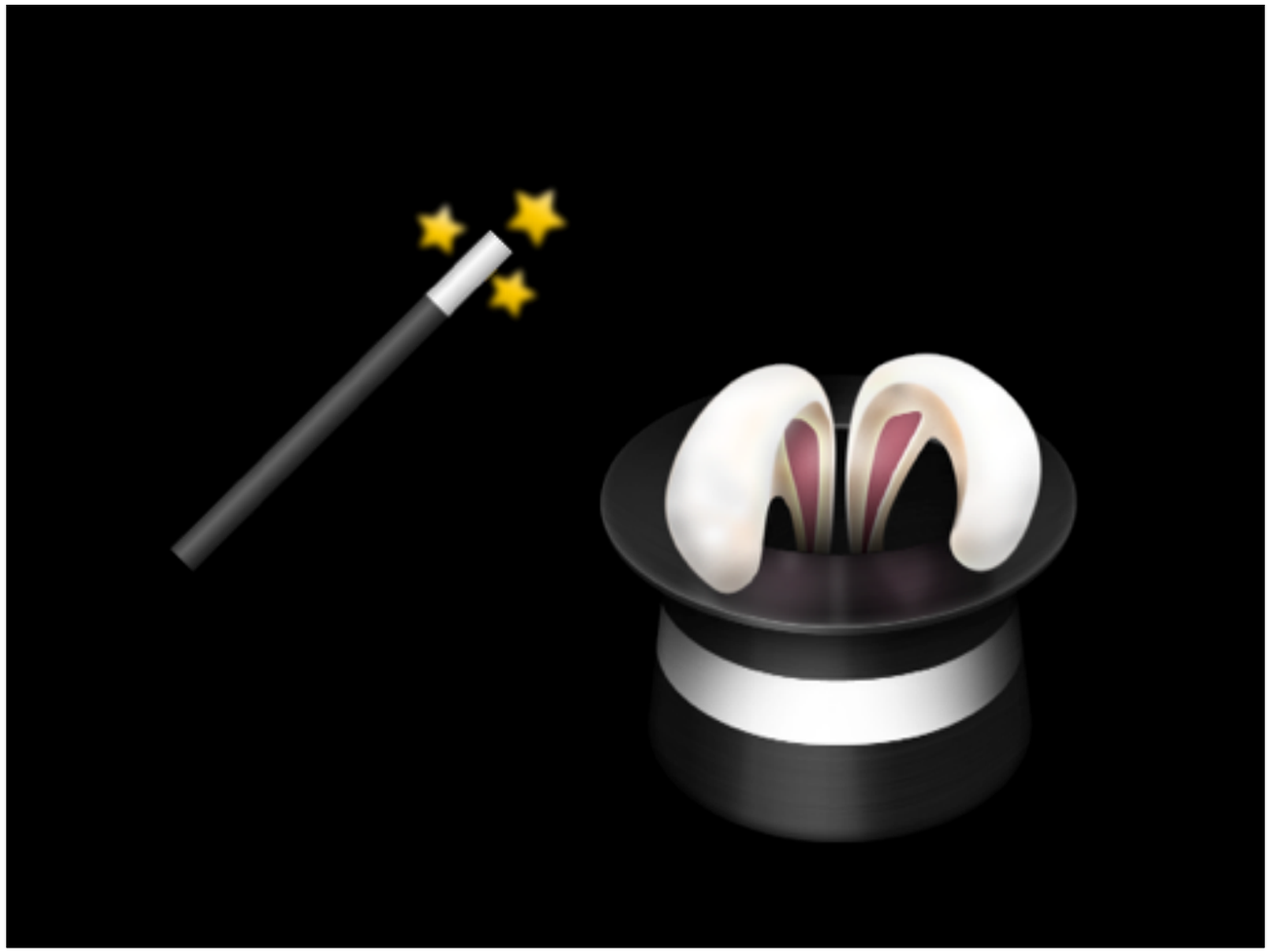
```
INSTALLED_APPS = (
    # …
    'rest_framework',
    'api',
)
```


127.0.0.1:8000/api/
Django REST framework v 2.4.3
Api Root
Api Root
OPTIONS
GET
GET /api/
HTTP 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS
{
"v1/dools": "http://127.0.0.1:8000/api/v1/dools/"
}


127.0.0.1:8000/api/v1/deals/
Django REST framework v 2.4.3
Api Root · Deals List
Deals List
OPTIONS
GET
GET /api/v1/deals/
HTTP 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, POST, HEAD, OPTIONS
Raw data
HTML form
name:
value:
0
POST

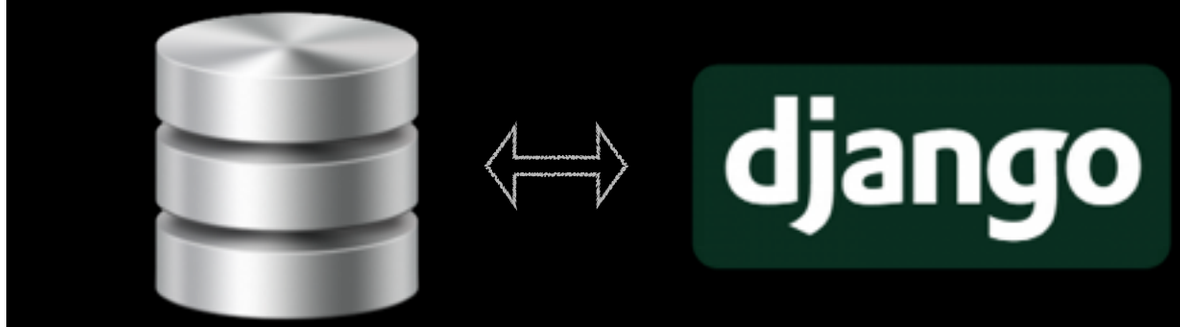

127.0.0.1:8000/api/v1/deals/
Django REST framework v 2.4.3
Api Root · Deals List
Deals List
OPTIONS
GET
POST /api/v1/deals/
HTTP 201 CREATED
Content-Type: application/json
Vary: Accept
Allow: GET, POST, HEAD, OPTIONS
{
"id": 1,
"name": "Amazing Web API",
"value": 42,
"updated_at": "2014-10-14T20:33:04.581Z"
}
Raw data
HTML form
name:
Amazing Web API
value:
42
POST

```
$ curl -i http://127.0.0.1:8000/api/v1/deals/
HTTP/1.0 200 OK
Date: Tue, 14 Oct 2014 20:35:56 GMT
Server: WSGIServer/0.1 Python/2.7.8
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN
Content-Type: application/json
Allow: GET, POST, HEAD, OPTIONS

[{"id": 1, "name": "Amazing Web API", "value": 42,
"updated_at": "2014-10-14T20:33:04.581Z"}]
```

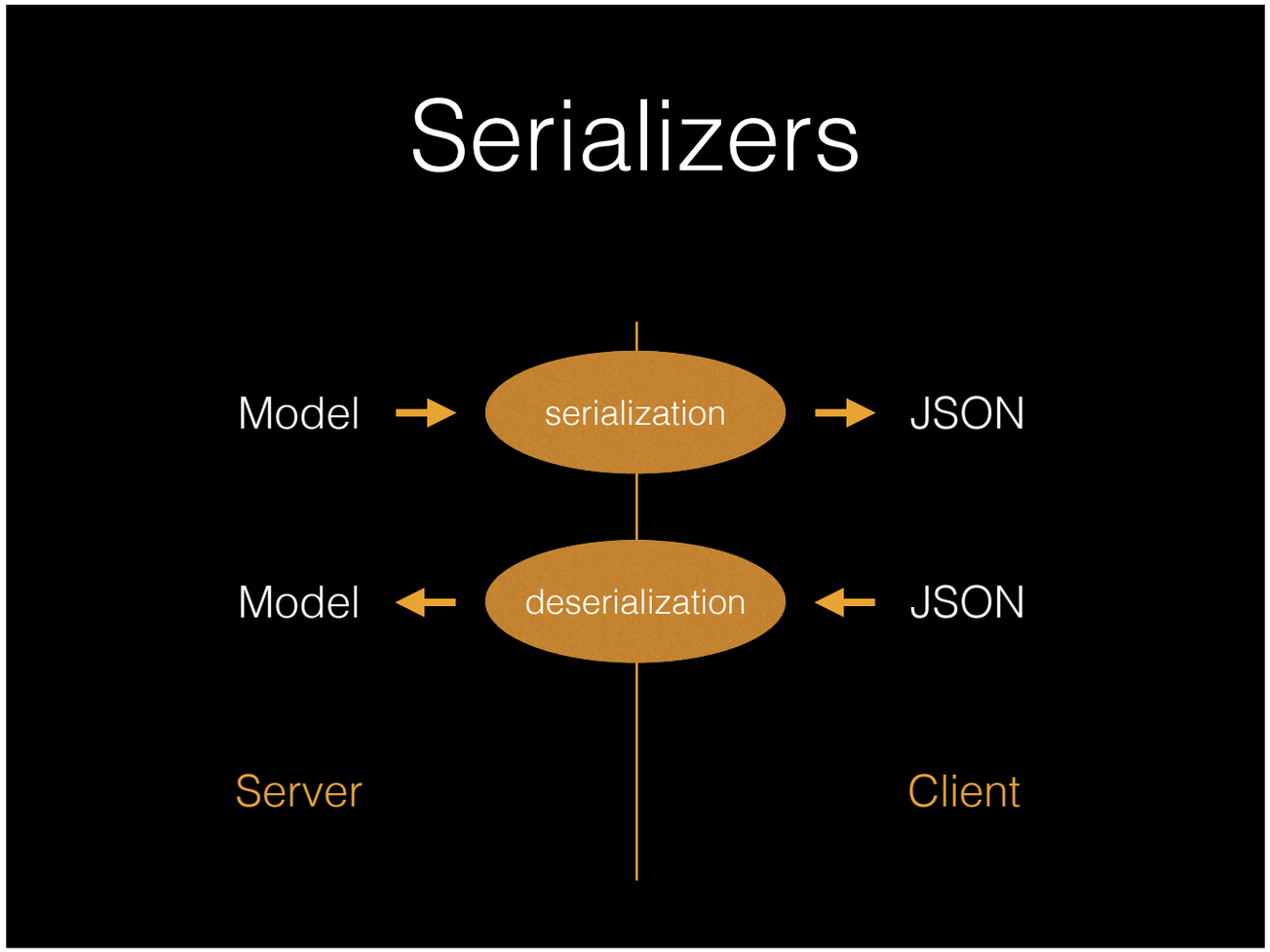


Magic, ahn? Okay, now let’s understand it in detail.

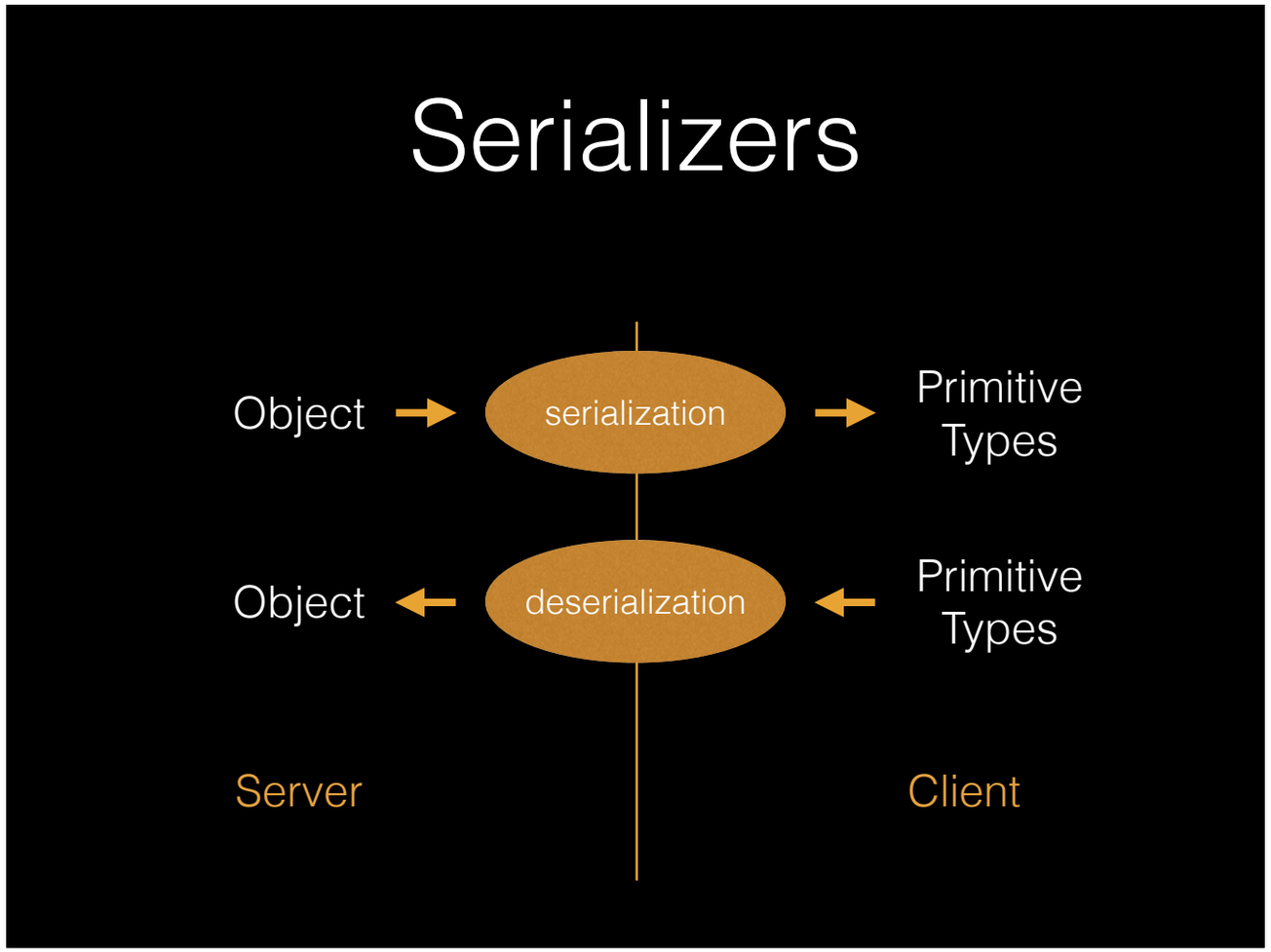
Models



The model is where the mapping between your database and Python/Django lives. There's nothing special about DRF here, it's just plain Django.



In order to expose your model as JSON, bidirectionally, you need a serializer. The serializer plays a role of a translator from a Model to JSON and vice-versa.

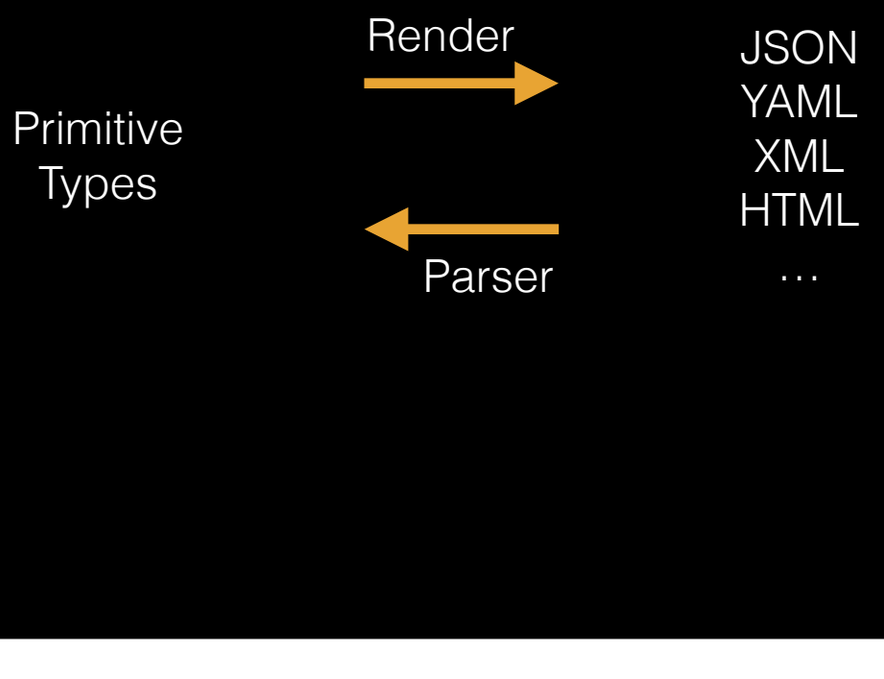


In reality, the serializer is concerned with converting arbitrary Python objects into Python primitive types (dict, list, string, integer) and vice-versa. DRF provides us ModelSerializers for solving the common case when those objects are Models. But we have full control of how serialization works, if we want to.

```python
class DealSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(max_length=255)
    value = serializers.IntegerField()
    updated_at = serializers.DateTimeField(read_only=True)

    def restore_object(self, attrs, instance=None):
        if instance is not None:
            instance.name = attrs.get('name', instance.name)
            instance.value = attrs.get('value', instance.value)
            return instance
        return Deal(**attrs)
```

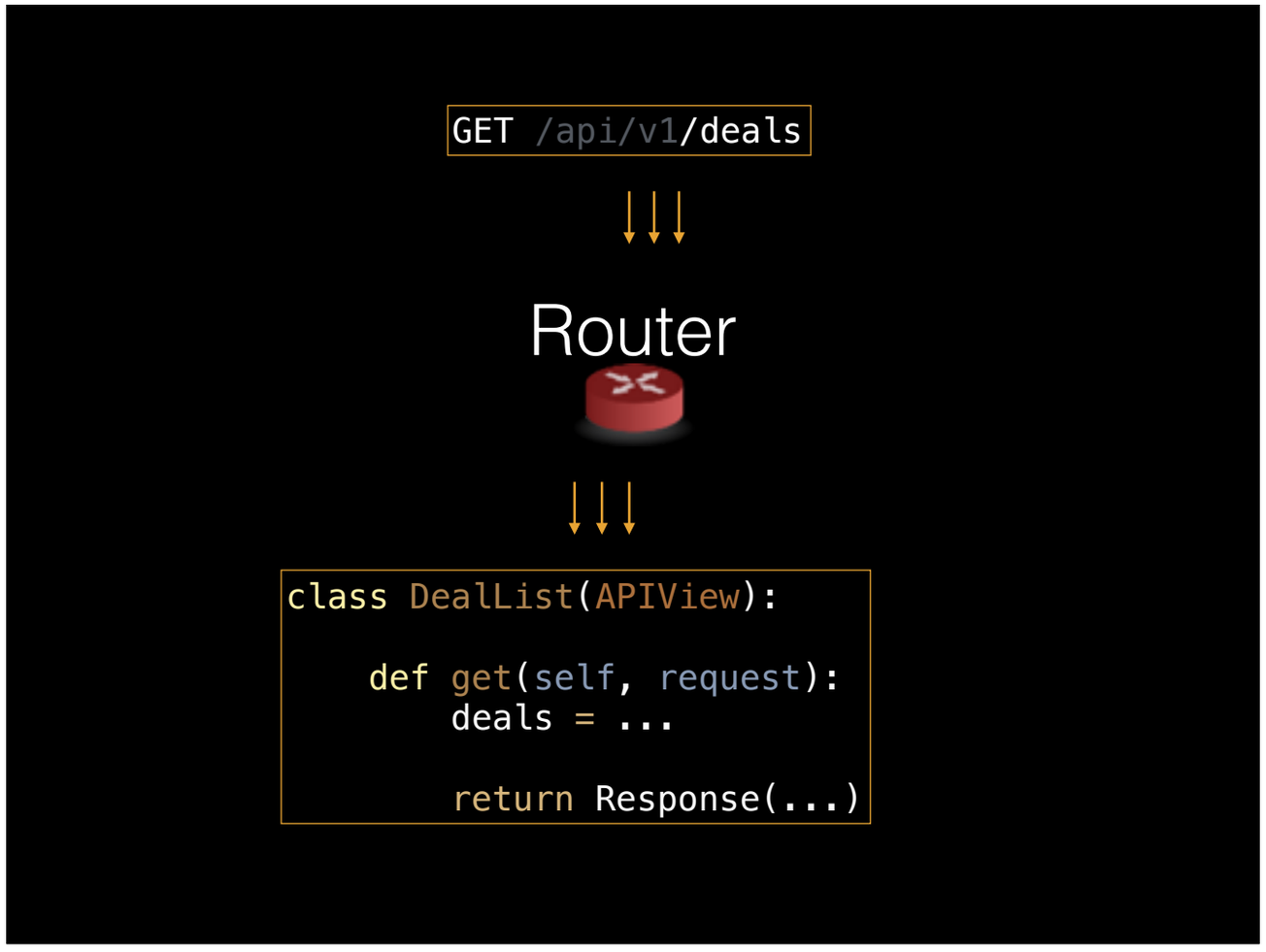
Here's how we can implement the same functionality provided by the ModelSerializer class we saw before. DRF gives us total flexibility to implement our own custom serializers. We declare fields and how to go from Python primitives into a specialized object. The syntax is very similar to Django Forms and Models.

Content Negotiation



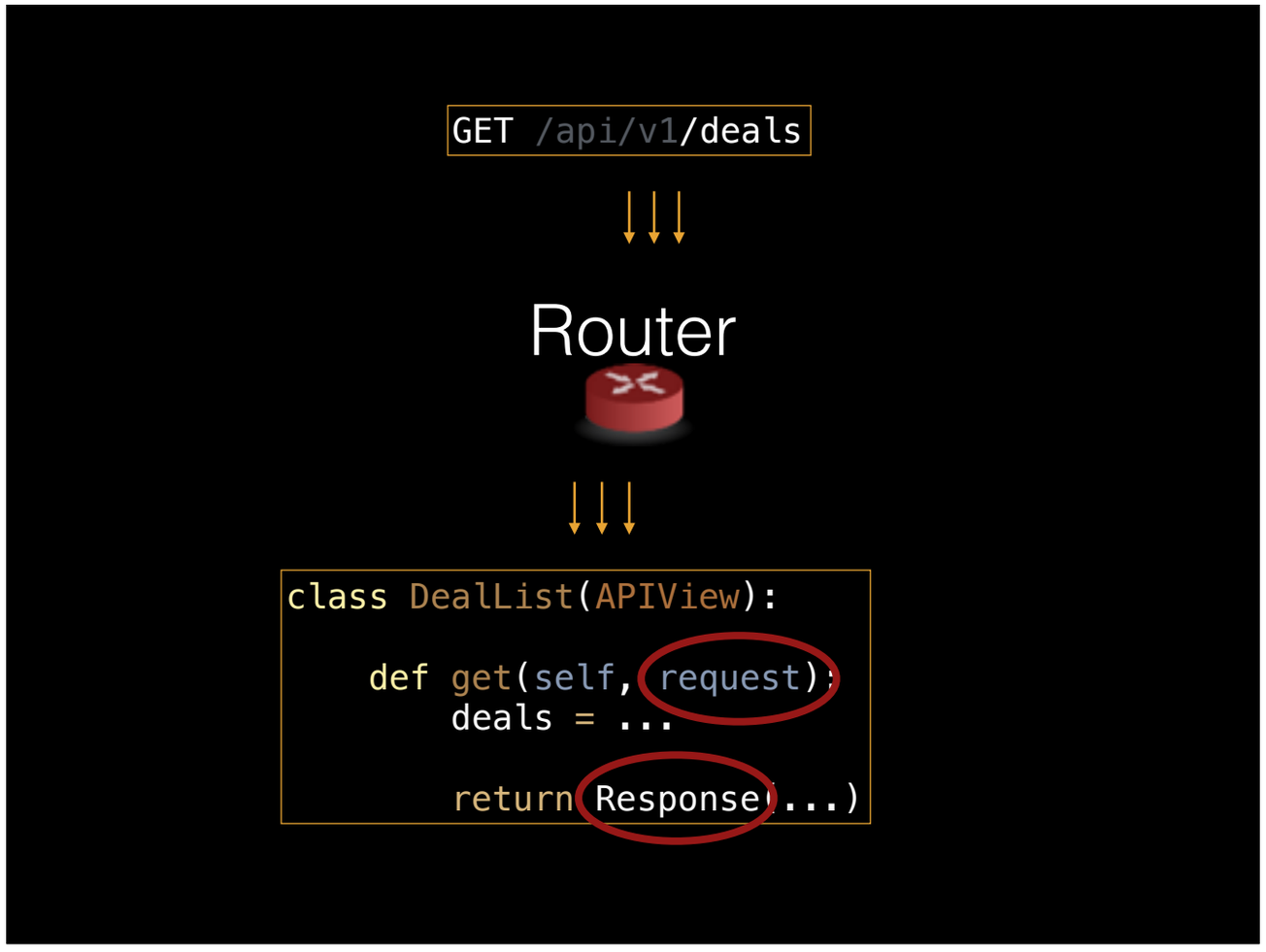
We can also customize how Content Negotiation works, and here's where primitive Python types are converted to/from JSON or other formats. The Browsable API interface is one of those formats.

Now let’s zoom out and look how a HTTP request eventually results in some JSON.



The router is an abstraction that facilitates creating URL bindings between URLs (API endpoints) and views. DRF provides specialized deal classes that handle content negotiation, authentication, permission, etc.

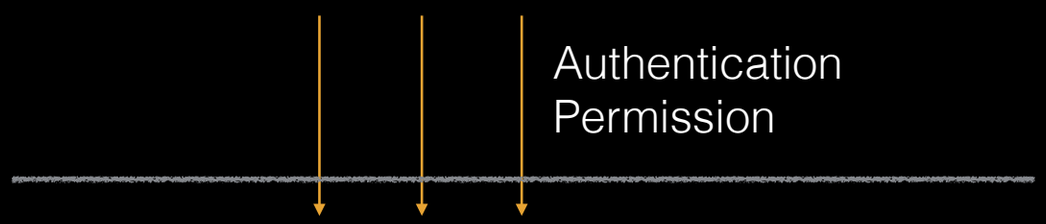
It’s the View that takes a request object as input and outputs a response object.



The router is an abstraction that facilitates creating URL bindings between URLs (API endpoints) and views. DRF provides specialized deal classes that handle content negotiation, authentication, permission, etc.

It’s the View that takes a request object as input and outputs a response object.

Router




```
class DealList(APIView):
    authentication_classes = (TokenAuthentication,)
    permission_classes = (IsAdminUser,)

    def get(self, request):
        deals = ...
        return Response(...)
```

You can specify authentication and permission details for your API either globally in your settings.py, or in each view. Those act as middleware that are applied before the request object actually gets into the view method to handle the request.

Authentication

```python
class CustomAuthentication(BaseAuthentication):

    def authenticate(self, request):

        token = request.META.get("HTTP_X_TOKEN")
        if not token:
            return None

        try:
            user = get_user_by_token(token)
        except UserNotFound:
            raise exceptions.AuthenticationFailed

        return (user, "custom_auth")
```

```python
class CustomAuthentication(BaseAuthentication):

    def authenticate(self, request):

        token = request.META.get("HTTP_X_TOKEN")
        if not token:
            return None

        try:
            user = get_user_by_token(token)
        except UserNotFound:
            raise exceptions.AuthenticationFailed

        return (user, "custom_auth")
```

```python
class CustomAuthentication(BaseAuthentication):

    def authenticate(self, request):

        token = request.META.get("HTTP_X_TOKEN")
        if not token:
            return None

        try:
            user = get_user_by_token(token)
        except UserNotFound:
            raise exceptions.AuthenticationFailed

        return (user, "custom_auth")
```

```python
class CustomAuthentication(BaseAuthentication):

    def authenticate(self, request):

        token = request.META.get("HTTP_X_TOKEN")
        if not token:
            return None

        try:
            user = get_user_by_token(token)
        except UserNotFound:
            raise exceptions.AuthenticationFailed

        return (user, "custom_auth")
```

```python
class CustomAuthentication(BaseAuthentication):

    def authenticate(self, request):

        token = request.META.get("HTTP_X_TOKEN")
        if not token:
            return None

        try:
            user = get_user_by_token(token)
        except UserNotFound:
            raise exceptions.AuthenticationFailed

        return (user, "custom_auth")
```

```python
class CustomAuthentication(BaseAuthentication):

    def authenticate(self, request):

        token = request.META.get("HTTP_X_TOKEN")
        if not token:
            return None

        try:
            user = get_user_by_token(token)
        except UserNotFound:
            raise exceptions.AuthenticationFailed

        return (user, "custom_auth")
```

```python
class CustomAuthentication(BaseAuthentication):

    def authenticate(self, request):

        token = request.META.get("HTTP_X_TOKEN")
        if not token:
            return None

        try:
            user = get_user_by_token(token)
        except UserNotFound:
            raise exceptions.AuthenticationFailed

        return (user, "custom_auth")
```

Permissions

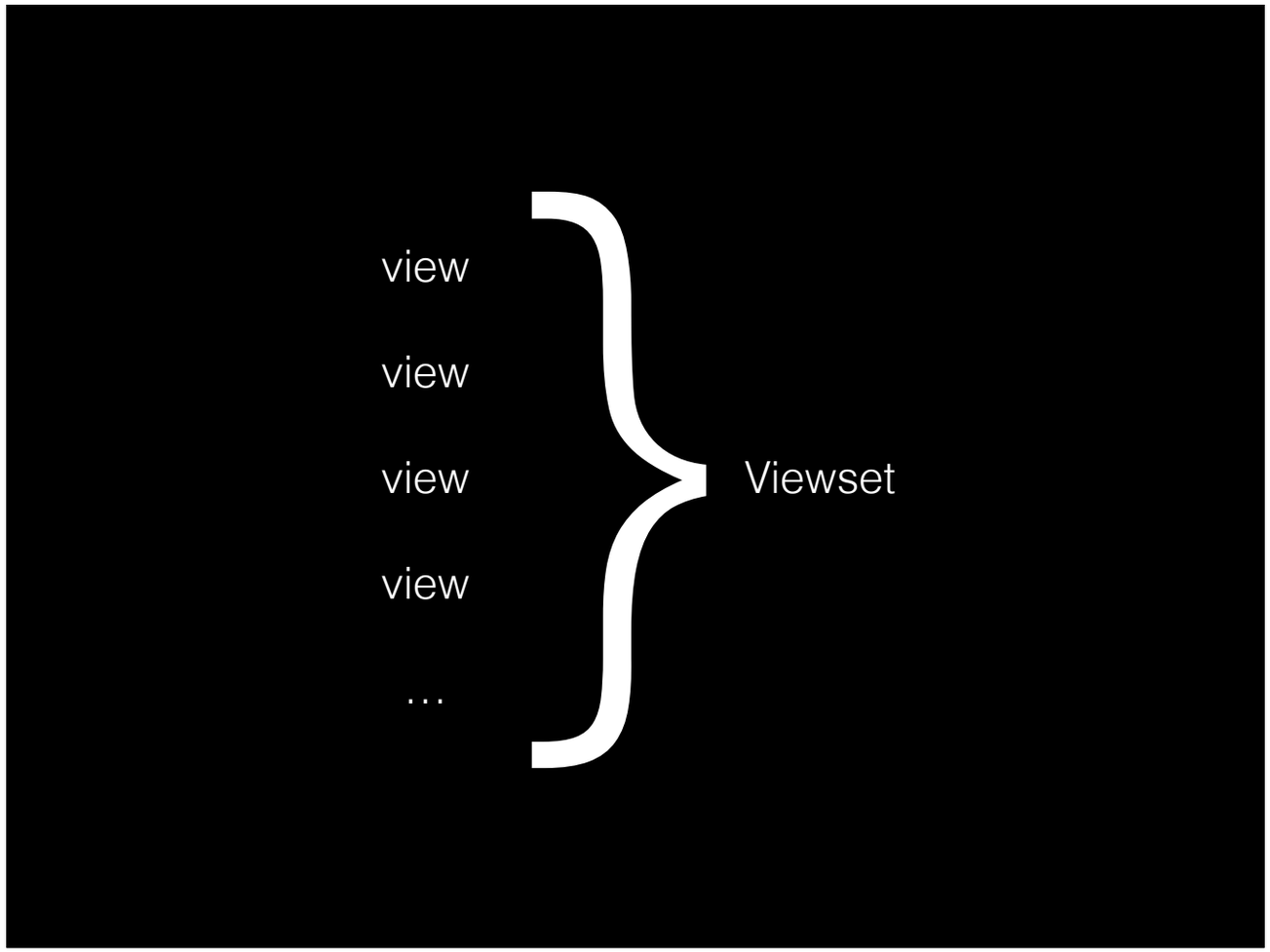
```python
class RandomAccess(BasePermission):
    def has_permission(self, request, view):
        return random.random() > 0.5


class IsOwner(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.owner == request.user
```

```python
class RandomAccess(BasePermission):
    def has_permission(self, request, view):
        return random.random() > 0.5


class IsOwner(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.owner == request.user
```

```python
class RandomAccess(BasePermission):
    def has_permission(self, request, view):
        return random.random() > 0.5


class IsOwner(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.owner == request.user
```

```python
class RandomAccess(BasePermission):
    def has_permission(self, request, view):
        return random.random() > 0.5


class IsOwner(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.owner == request.user
```

```python
class RandomAccess(BasePermission):
    def has_permission(self, request, view):
        return random.random() > 0.5


class IsOwner(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.owner == request.user
```



Instead of writing a lot of views separately, you normally have a set of views in your API that are closely related (list, create, update, delete). DRF gives you Viewsets that enable you to coherently implement all related code in a single place, or, even better, you might as well use generic viewsets and write less up to no code.
Viewsets integrate with routers.

```python
class DealViewSet(viewsets.ViewSet):

    def create(self, request):

        serializer = DealSerializer(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)
```

Here is where we finally glue it all together.

DRF requests' are a little different than Django's, with some little improvements like request.DATA and request.QUERY_PARAMS. The Response class subclasses Django's SimpleTemplateResponse.

You'll use your serializer both to get data out of the request and to send data down through the response.

```python
class DealViewSet(viewsets.ViewSet):

    def create(self, request):

        serializer = DealSerializer(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)
```

Here is where we finally glue it all together.

DRF requests' are a little different than Django's, with some little improvements like request.DATA and request.QUERY_PARAMS. The Response class subclasses Django's SimpleTemplateResponse.

You'll use your serializer both to get data out of the request and to send data down through the response.

```python
class DealViewSet(viewsets.ViewSet):

    def create(self, request):

        serializer = DealSerializer(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)
```

Here is where we finally glue it all together.

DRF requests' are a little different than Django's, with some little improvements like request.DATA and request.QUERY_PARAMS. The Response class subclasses Django's SimpleTemplateResponse.

You'll use your serializer both to get data out of the request and to send data down through the response.

```
class DealViewSet(viewsets.ViewSet):

    def create(self, request):

        serializer = DealSerializer(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)
```

Here is where we finally glue it all together.

DRF requests' are a little different than Django's, with some little improvements like request.DATA and request.QUERY_PARAMS. The Response class subclasses Django's SimpleTemplateResponse.

You'll use your serializer both to get data out of the request and to send data down through the response.

Next Steps

- Testing
- Tune Django
- Build your own patterns
- API Throttling, Filtering
- Auto generate API docs

So we've gone through the main concepts in DRF. This was a lot of content to pack in so little time, and there are a bunch of interesting things I didn't have a change to mention.

I wish you were able to grasp the main points, this was just the beginning of a conversation. Let's talk.

What are your questions?

Thank you

Rodolfo Carvalho
Python Lead Developer, Base Lab

rodolfo@getbase.com
rodolfocarvalho.net
py.getbase.com