Django pt. 4: Rest Framework & APIs

Kianoosh Abbasi

CSC309 Winter 2022

So far

- How web works, HTML, CSS
- Django intro
  Setup, simple views, forms, templates
- MVC, database, ORM, Auth
- Custom models, migrations, class-based views

This session

- More on CRUD views
- Restful APIs
  VERY IMPORTANT: Project phase 2 is all about it!
- JSON
- APIViews, serializers, permissions

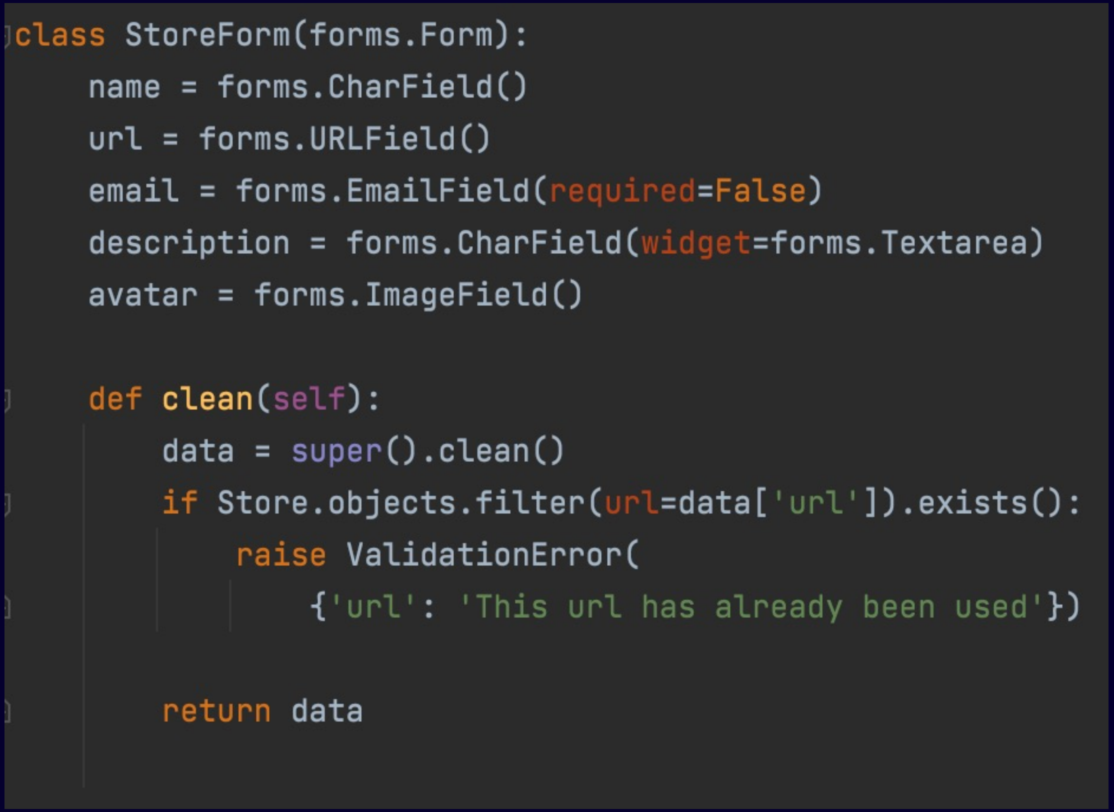
FormView

- A way to organize forms
  Separates the form logic from the view logic

- Form class:
  Define fields one by one
  Define a clean method for validation

- FormView:
  Specify `form_class` attribute
  Specify `success_url` or `get_success_url`
  Override `form_valid` to apply the changes

forms/store_form.py

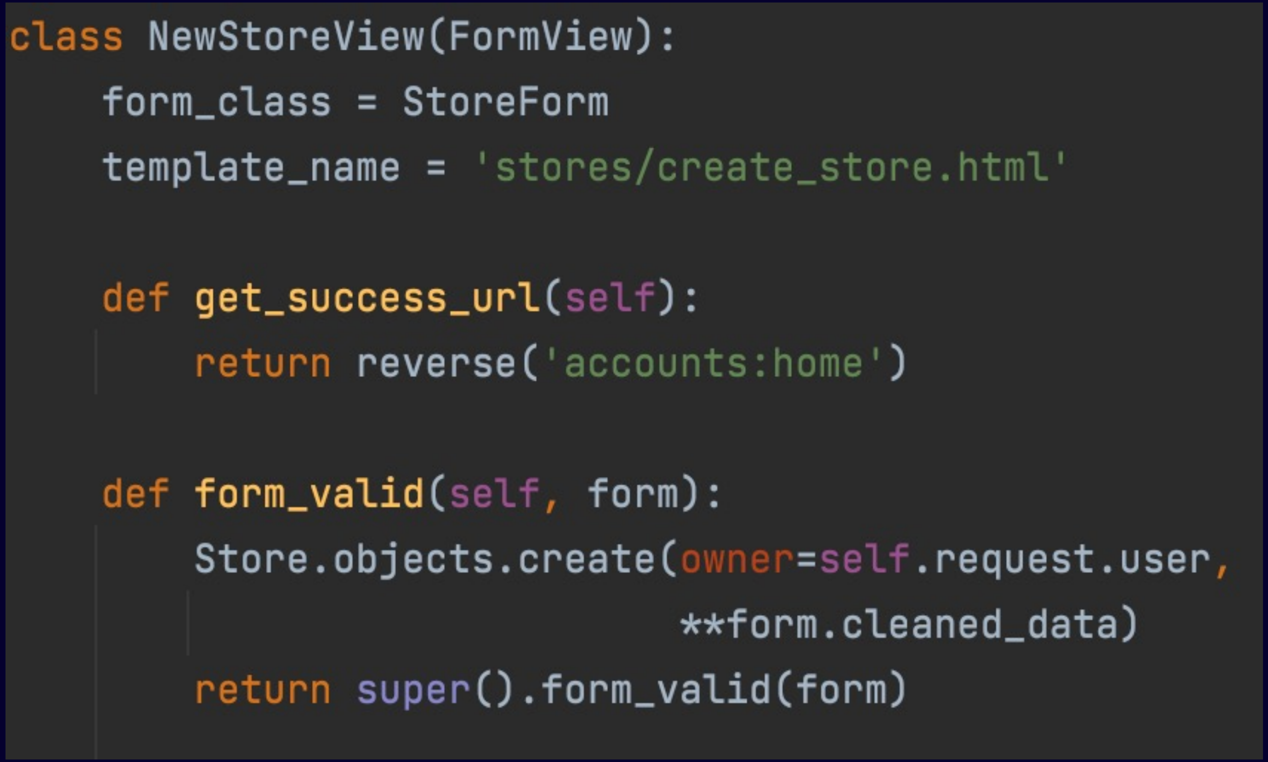


```python
class StoreForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    email = forms.EmailField(required=False)
    description = forms.CharField(widget=forms.Textarea)
    avatar = forms.ImageField()

    def clean(self):
        data = super().clean()
        if Store.objects.filter(url=data['url']).exists():
            raise ValidationError(
                {'url': 'This url has already been used'})

        return data
```

views/new_store.py



```python
class NewStoreView(FormView):
    form_class = StoreForm
    template_name = 'stores/create_store.html'

    def get_success_url(self):
        return reverse('accounts:home')

    def form_valid(self, form):
        Store.objects.create(owner=self.request.user,
                             **form.cleaned_data)
        return super().form_valid(form)
```

Template

- The form instance is being created at every request
- GET request: `{{ form }}` sent to template context
- Up to developer to use it at all!
  You can stick with your html input tags (recommended)
- Reason: MVC's view should be separated from controller

- POST request: form instance created and populated with POST data

- Goes through validation and calls `form_valid` (redirect) or `form_invalid`

- `form_invalid` renders the template (like GET)

  This time, the `{{ form }}` instance has values and errors

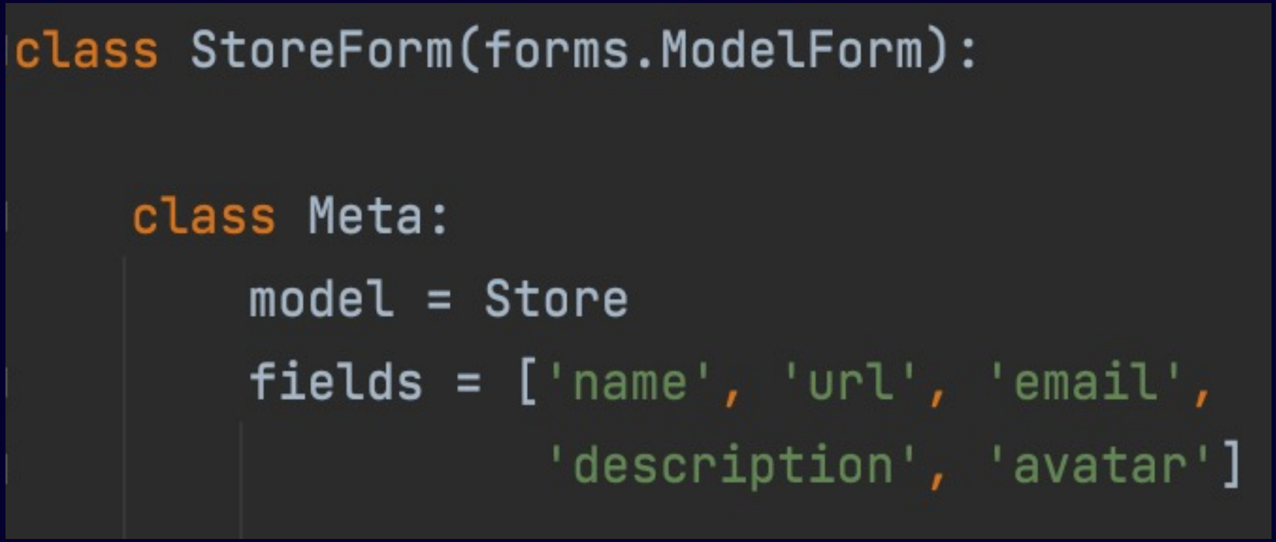
- Use values and errors at template

  ```
  {{ form.name }} {{ form.name.error }}
  {{ form.non_field_errors }}
  ```

ModelForm

- A shortcut to infer field types and validators from the model

- Addresses the coupling between models and forms

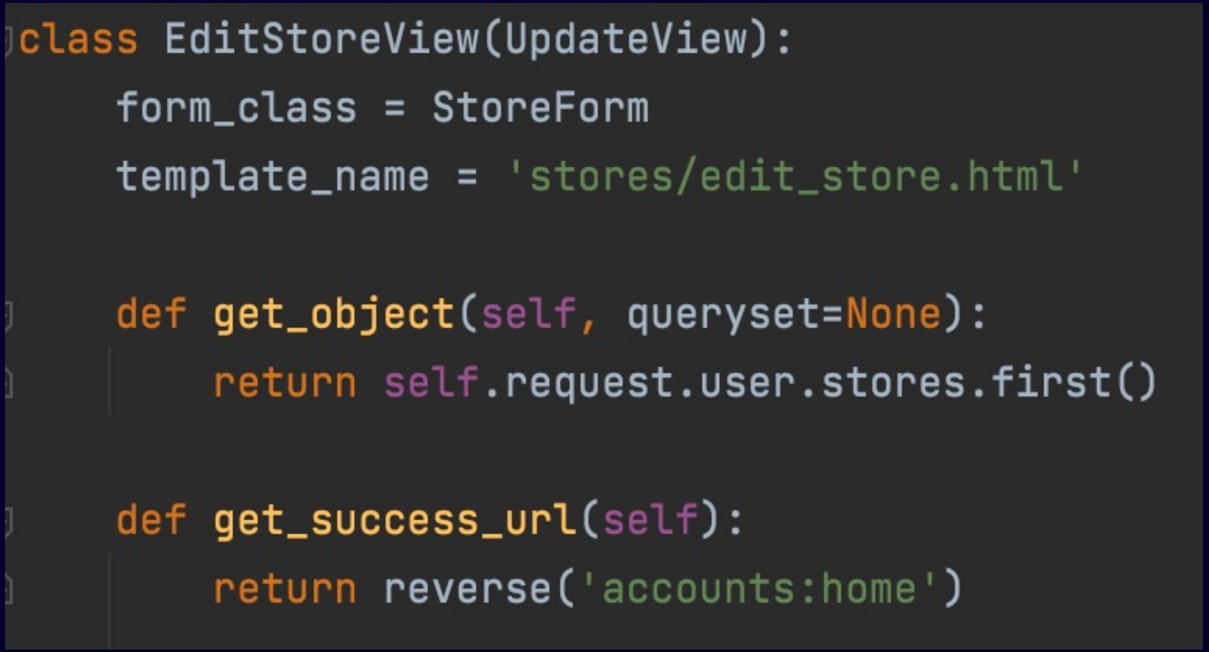


```python
class StoreForm(forms.ModelForm):

    class Meta:
        model = Store
        fields = ['name', 'url', 'email',
                  'description', 'avatar']
```

CreateView and UpdateView

- CreateView: a FormView whose `form_class` is a ModelForm
- UpdateView: a CreateView that implements `get_object`
- `form_valid` is already implemented and may be useful!



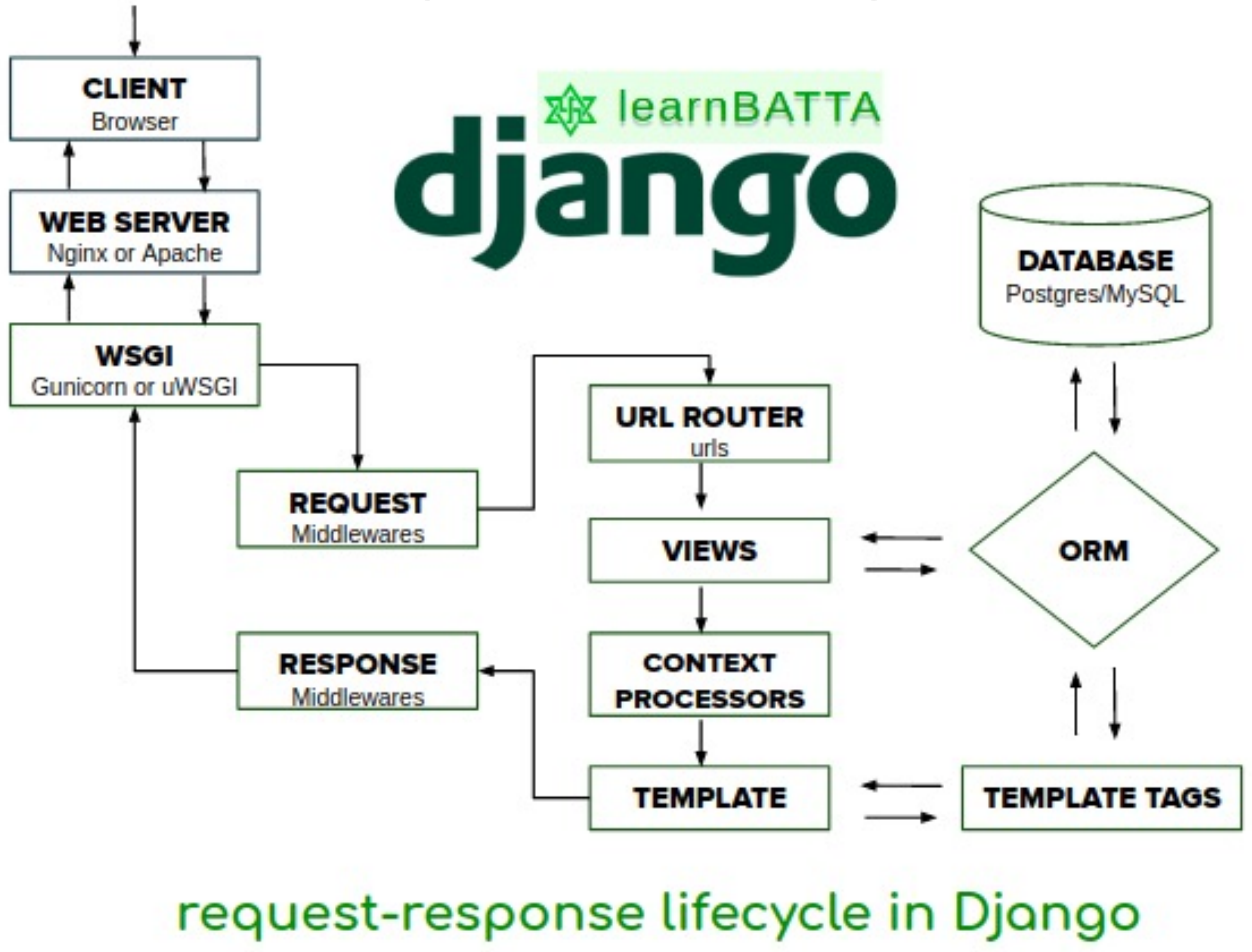
```
class EditStoreView(UpdateView):
    form_class = StoreForm
    template_name = 'stores/edit_store.html'

    def get_object(self, queryset=None):
        return self.request.user.stores.first()

    def get_success_url(self):
        return reverse('accounts:home')
```

Restful APIs

Current way of building a website



Source: https://learnbatta.com/blog/understanding-request-response-lifecycle-in-django-29/

Caveats?

- Too backend-oriented
  All frontend logic is served as static files
- Django is backend framework but contains frontend codes
- Backend and frontend in the same place
  Can’t use a dedicated frontend framework like React
- Frontend can’t be as sophisticated
  Example: Single-page application

Why Separate backend and frontend?

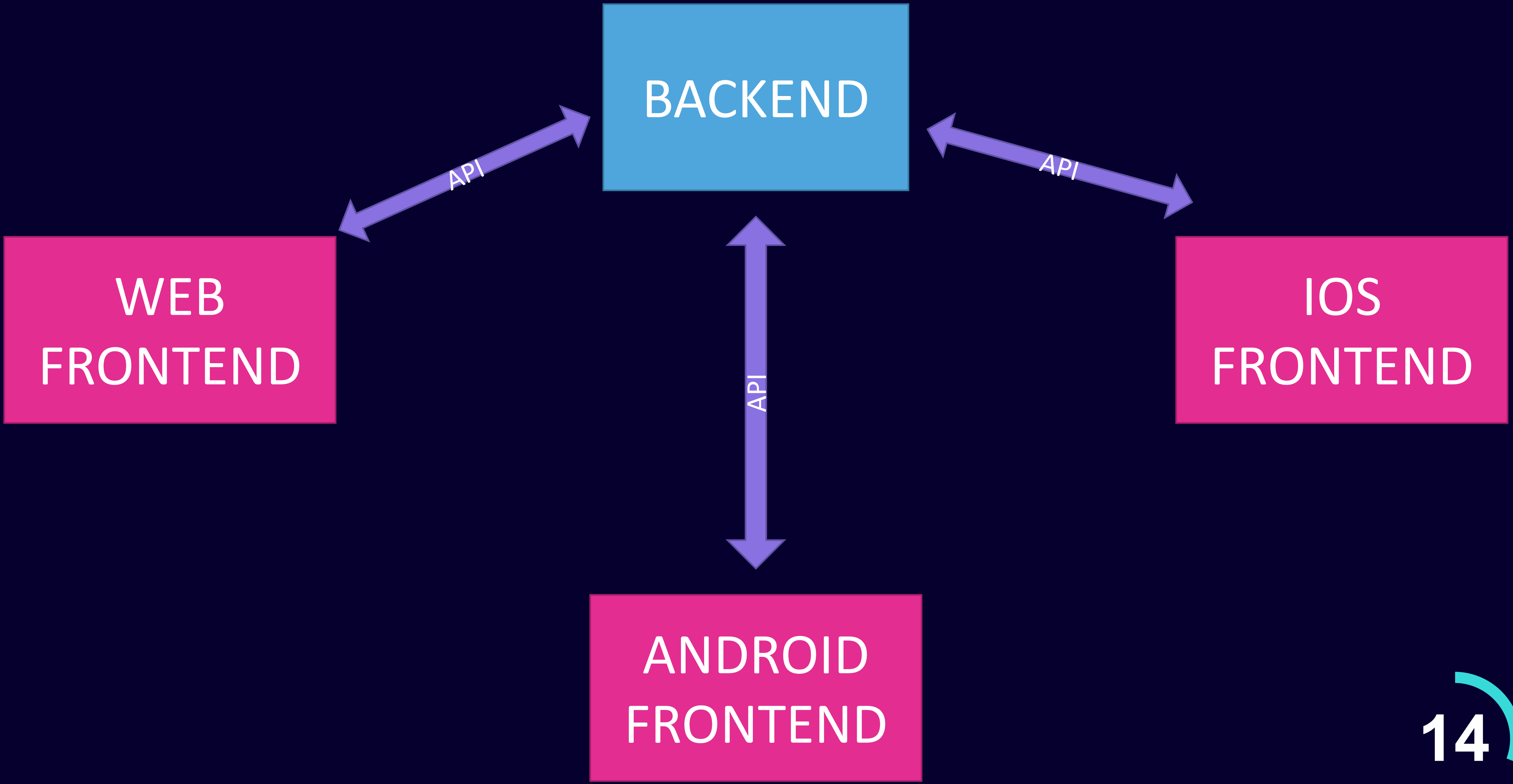
- Big projects are too big to contain both

- Modularity

  Changes in frontend will not affect backend and vice versa

- Consolidation

  One backend and multiple frontends (web, android, iOS)


BACKEND
API
API
API
WEB FRONTEND
IOS FRONTEND
ANDROID FRONTEND

Modularity

- Different services/apps talk to each other with a protocol

- API: The way an application can be talked to
  - Stands for Application Programming Interface

- Web applications: typically, a set of HTTP requests

Separate Backend and Frontend

- Backend views are only about data retrieval and manipulation
- Backend does not care about how data is shown, UI, or UX
- No templates, no static files

Response format

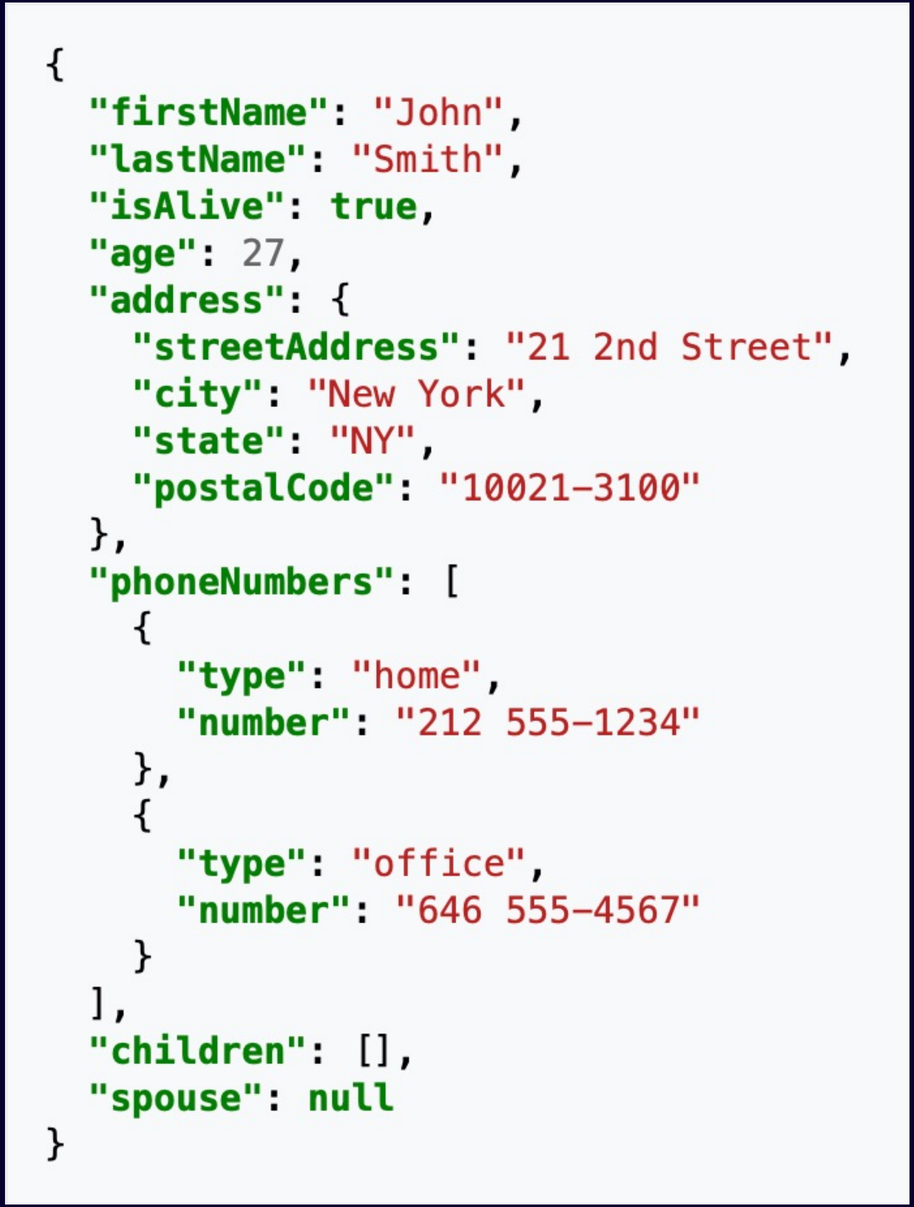
- HTML does not make sense anymore! WHY?

- A popular standard is JavaScript Object Notation or JSON
  - Derived from Javascript syntax for defining objects

- Easy, human-readable, and fast
  - Many languages (python, javascript, …) have built-in parsers and support

JSON

- Primitive types: number, string, boolean, null
- Array: ordered collection of elements
- Object: key-value pairs
  - Keys are strings
- Array elements and object values can be of any type (string, null, array, object, etc.)



```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Source: wikipedia

So far this session

- Big projects may require a separation of backend and frontend

- Communication done through APIs
  - Data retrieval and manipulation

- Request/response follows JSON format

API architecture

- Representational State Transfer (REST)
- A set of URL endpoints that typically do a CRUD function
- Example: https://binancedocs.github.io/apidocs/spot/en

Django REST framework (DRF)

Django REST framework

- Makes writing Restful APIs easier

- Pre-written JSON parser, CRUD views, permissions, and serializers

- Still a Django project
  - Same models, urls, etc
  - Views are subclasses of DRF views

Setup

- Install via pip

  `pip install djangorestframework`

- Add `'rest_framework'` to `INSTALLED_APPS`

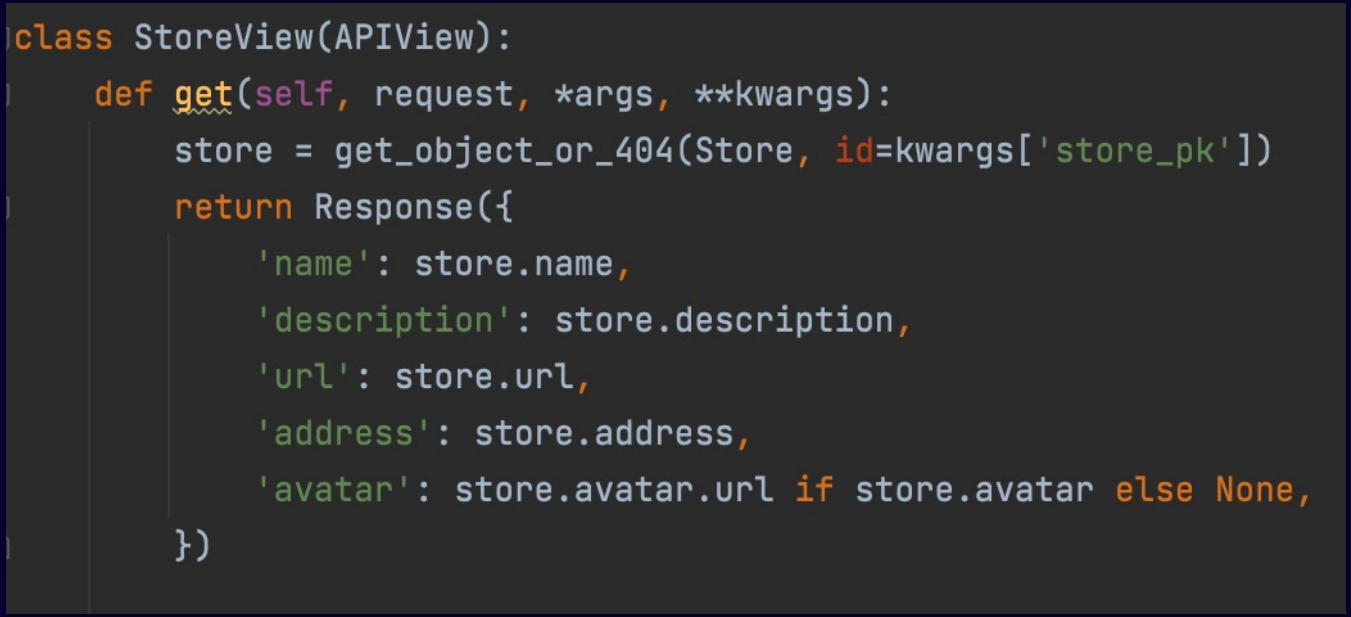
- There is no front-end:

  Install Postman to test APIs

  Use DRF’s browsable APIs at development

APIView

- Subclass of View
- Response gets a dictionary and returns an HTTP JSON reponse



```
class StoreView(APIView):
    def get(self, request, *args, **kwargs):
        store = get_object_or_404(Store, id=kwargs['store_pk'])
        return Response({
            'name': store.name,
            'description': store.description,
            'url': store.url,
            'address': store.address,
            'avatar': store.avatar.url if store.avatar else None,
        })
```

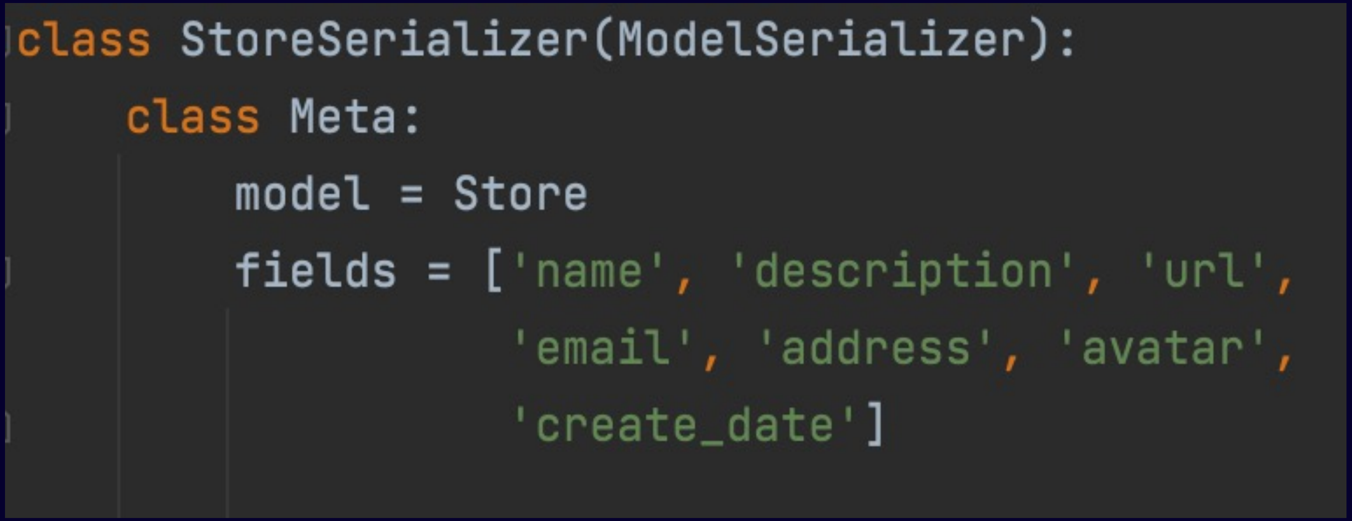
Generic (CRUD) Views

- Rest framework CRUD Views:
  ```
  CreateAPIView
  ListAPIView, RetrieveAPIView
  UpdateAPIView
  DestroyAPIView
  ```
- Override `create`, `list`, `retrieve`, `update`, `destroy` (respectively)
- Need to implement a serializer

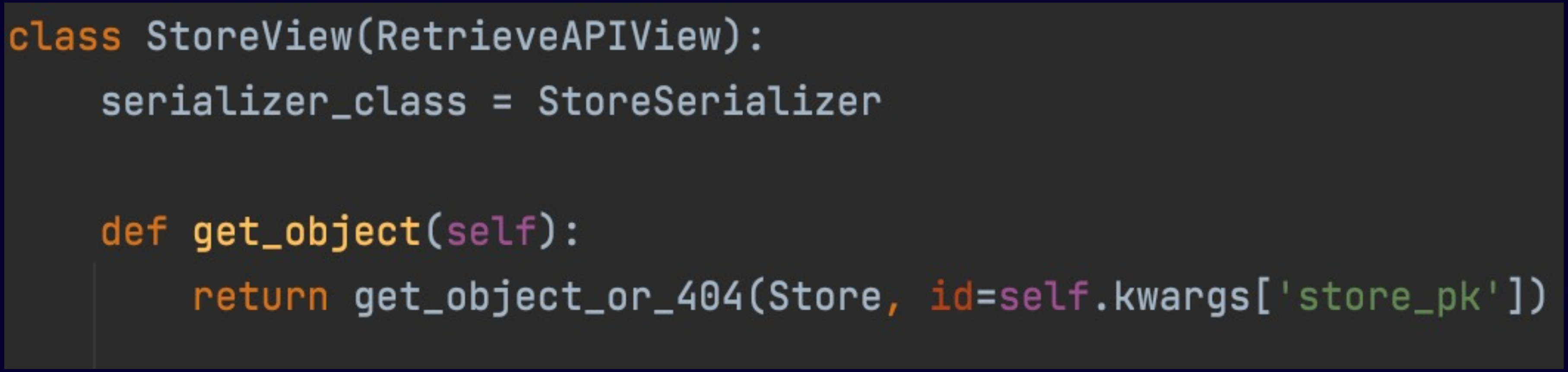
Serializer

- Model instances need to be serialized and deserialized for the end user
- Object is represented in a format that can be transferred or reconstructed later
- Dictionary (JSON) serializers in DRF
- Create *serializers.py* or *serializers* directory in your app

Serializers



```python
class StoreSerializer(ModelSerializer):
    class Meta:
        model = Store
        fields = ['name', 'description', 'url',
                  'email', 'address', 'avatar',
                  'create_date']
```



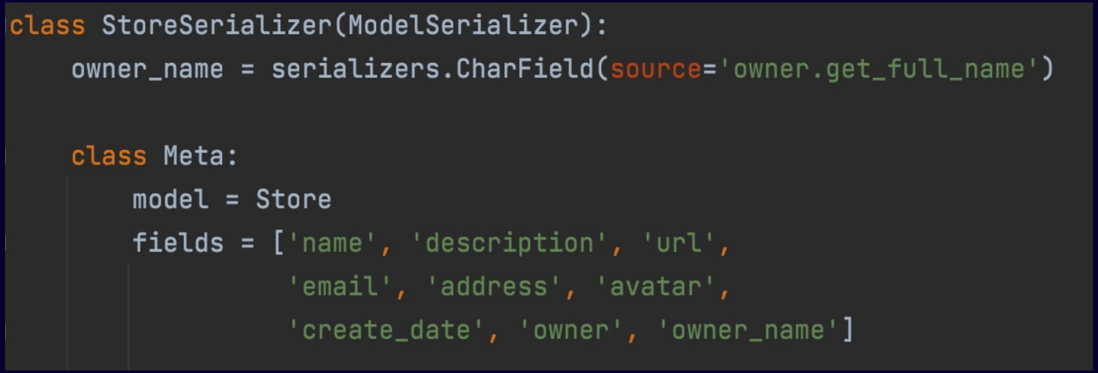
```python
class StoreView(RetrieveAPIView):
    serializer_class = StoreSerializer

    def get_object(self):
        return get_object_or_404(Store, id=self.kwargs['store_pk'])
```

More sophisticated fields

- Nested fields/properties can be accessed

- Should be defined in class body and then included in Meta fields



```
class StoreSerializer(ModelSerializer):
    owner_name = serializers.CharField(source='owner.get_full_name')

    class Meta:
        model = Store
        fields = ['name', 'description', 'url',
                  'email', 'address', 'avatar',
                  'create_date', 'owner', 'owner_name']
```

CRUD Views

- `ListAPIView`: Same as `RetrieveAPIView` but implements `get_queryset` instead
  - Same serializer can be used

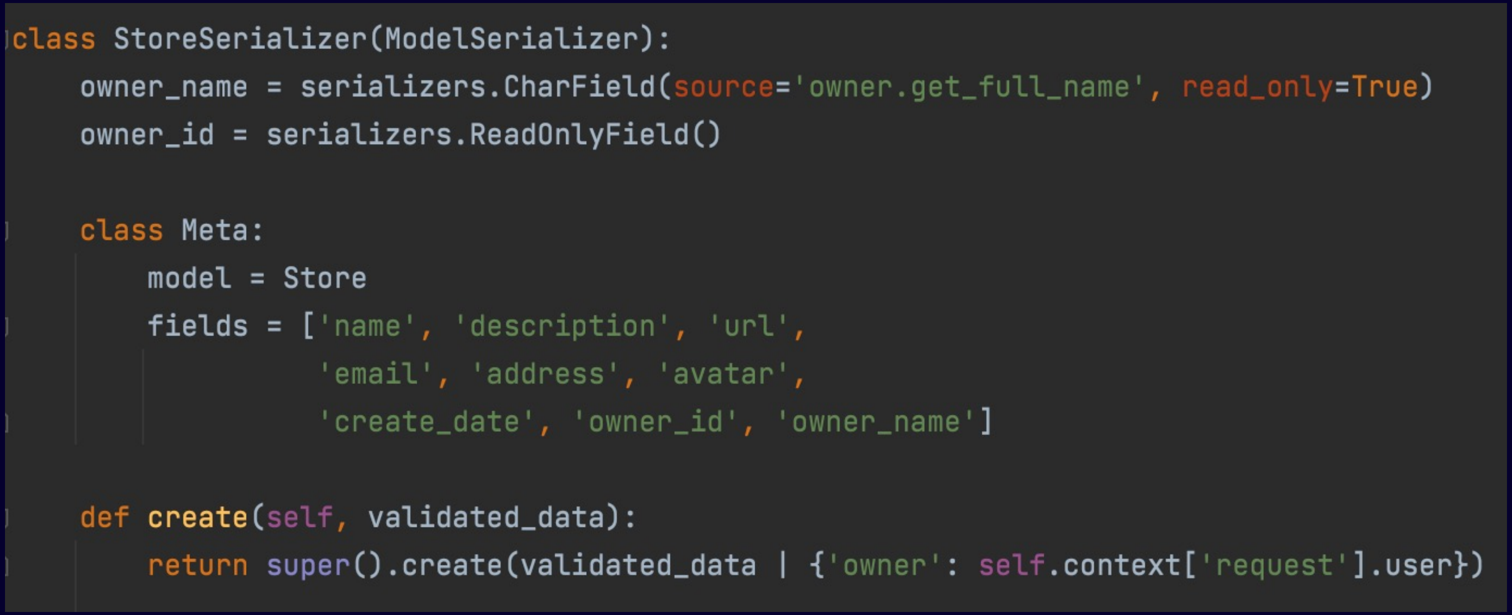
- Same serializers can be used to deserialize as well
  - `CreateAPIView` and `UpdateAPIView`
  - In some cases, you might need to create a separate serializer
  - All field validations are done automatically

Deserialization

- Goal is to use the same serializer
- Many fields are shared
- Exception: `owner_name` is not applicable anymore
- Exception: `owner` should be inferred from request.user at creation

Solution

- Certain fields can be defined as read_only (or write_only)
- request is passed through context



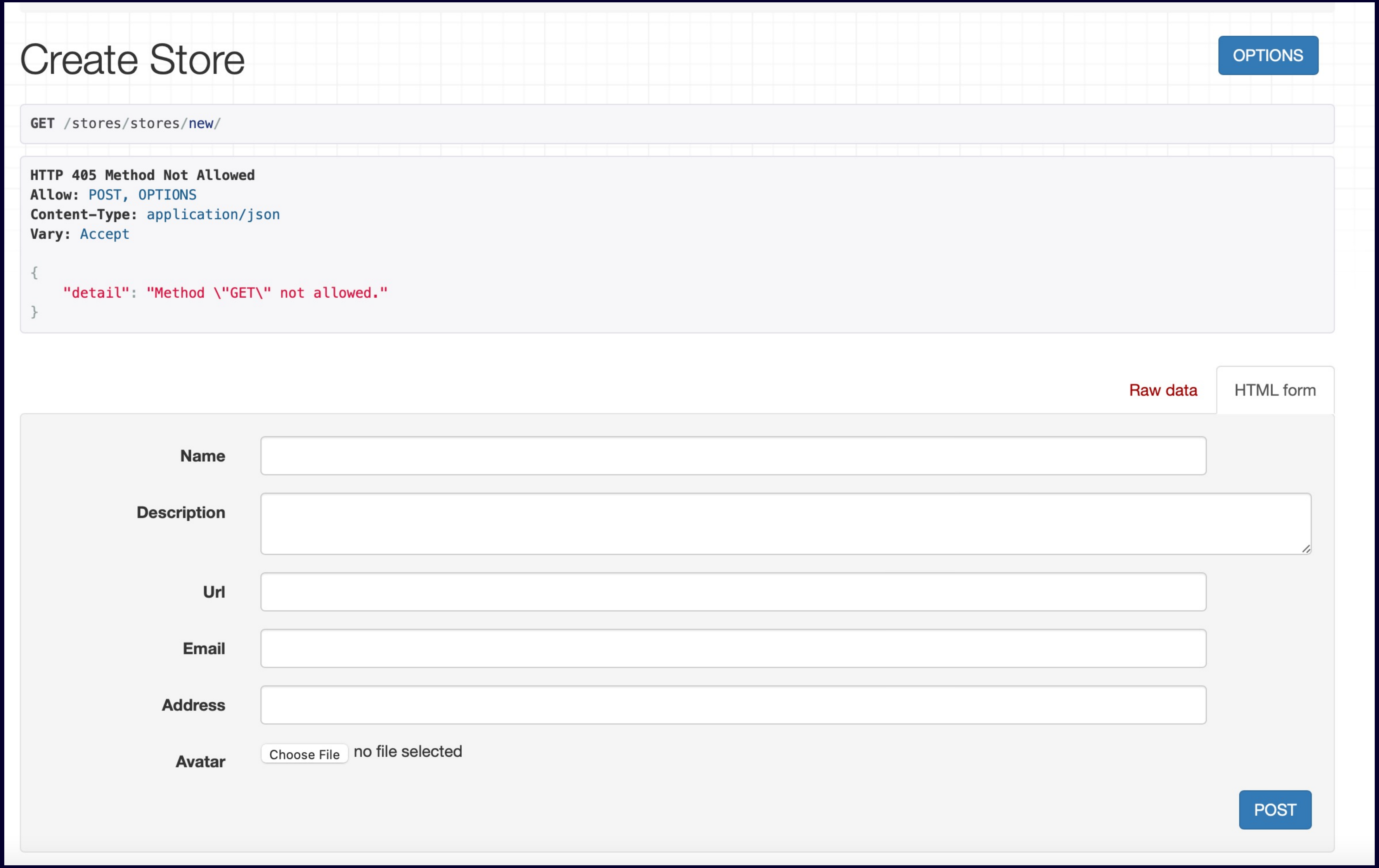
```
class StoreSerializer(ModelSerializer):
    owner_name = serializers.CharField(source='owner.get_full_name', read_only=True)
    owner_id = serializers.ReadOnlyField()

    class Meta:
        model = Store
        fields = ['name', 'description', 'url',
                  'email', 'address', 'avatar',
                  'create_date', 'owner_id', 'owner_name']

    def create(self, validated_data):
        return super().create(validated_data | {'owner': self.context['request'].user})
```

Browsable API

Perfect way to test your code during development



Create Store

```
GET /stores/stores/new/
```

```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Name	
Description	
Url	
Email	
Address	
Avatar	no file selected

Wait... what?

- Browsable API works with session auth, but there is no such a session in a real client

- Token auth is usually used instead

  Stateless tokens that are not stored in the server

- Install the JSON Web Token (JWT) package

  ```
  pip install djangorestframework-simplejwt
  ```

JWT Setup

Visit https://django-rest-framework-simplejwt.readthedocs.io/en/latest/getting_started.html

- Add JWT Auth to settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.BasicAuthentication',
    ),
}
```

- There is a default login view that generates a token

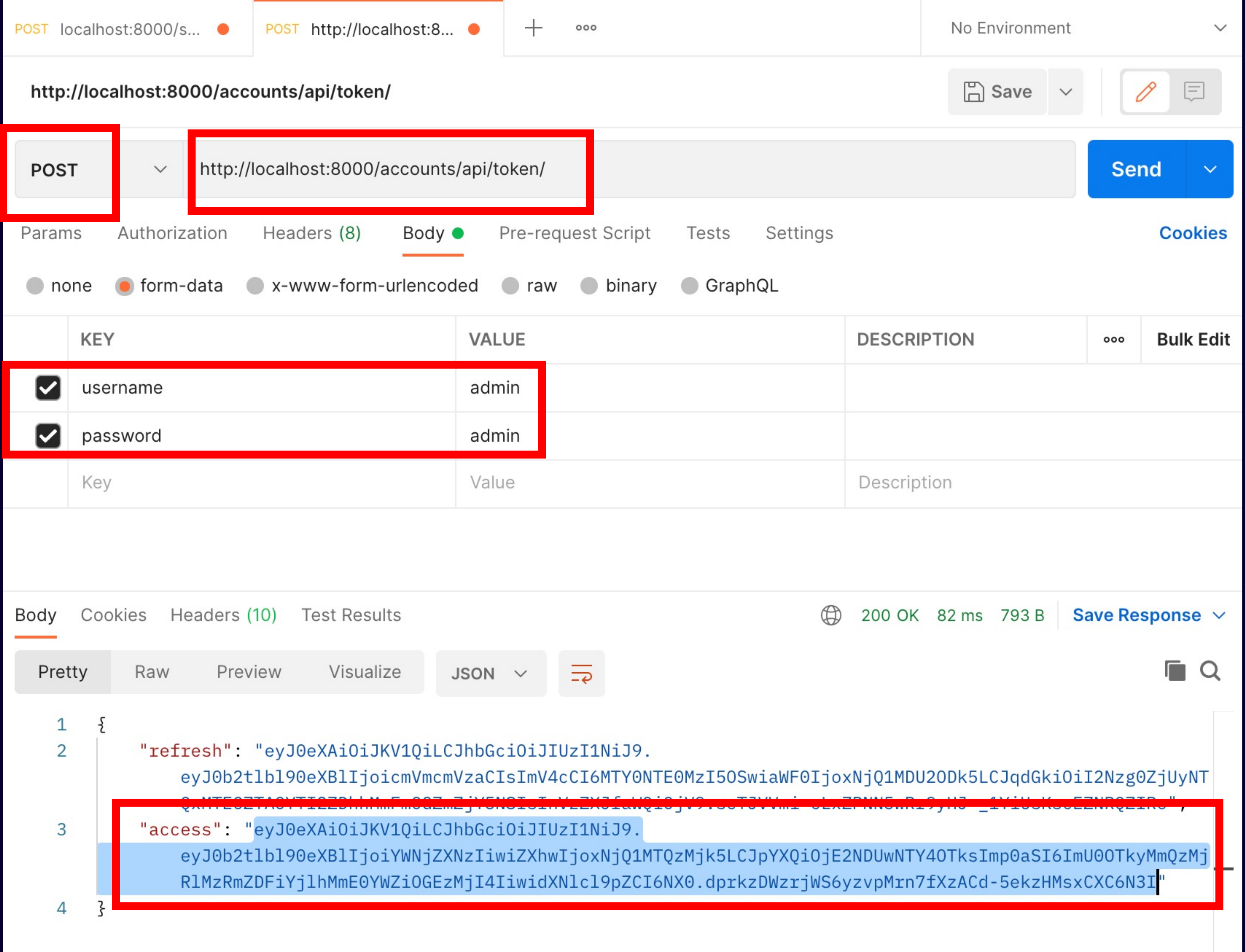
```
path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
```

- Token is short-lived (five minutes)

  Can be changed to other intervals in settings

  A refresh token can be used as well

Obtain a token with Postman



http://localhost:8000/accounts/api/token/

POST http://localhost:8000/accounts/api/token/

Params | Authorization | Headers (8) | Body | Pre-request Script | Tests | Settings

none [✓ form-data] x-www-form-urlencoded raw binary GraphQL

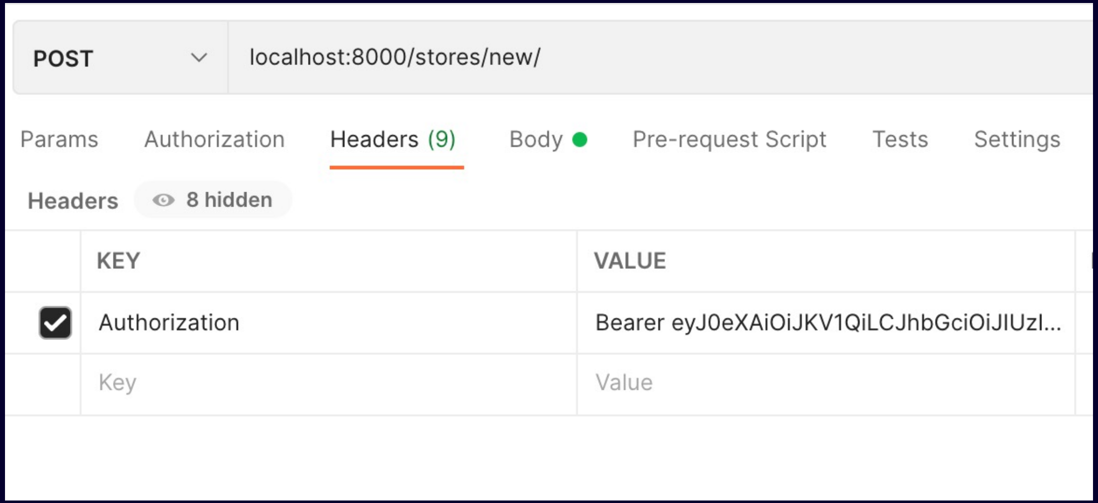
	KEY	VALUE	DESCRIPTION
✓	username	admin	
✓	password	admin	
	Key	Value	Description

Body | Cookies | Headers (10) | Test Results — 200 OK 82 ms 793 B — Save Response

Pretty | Raw | Preview | Visualize | JSON

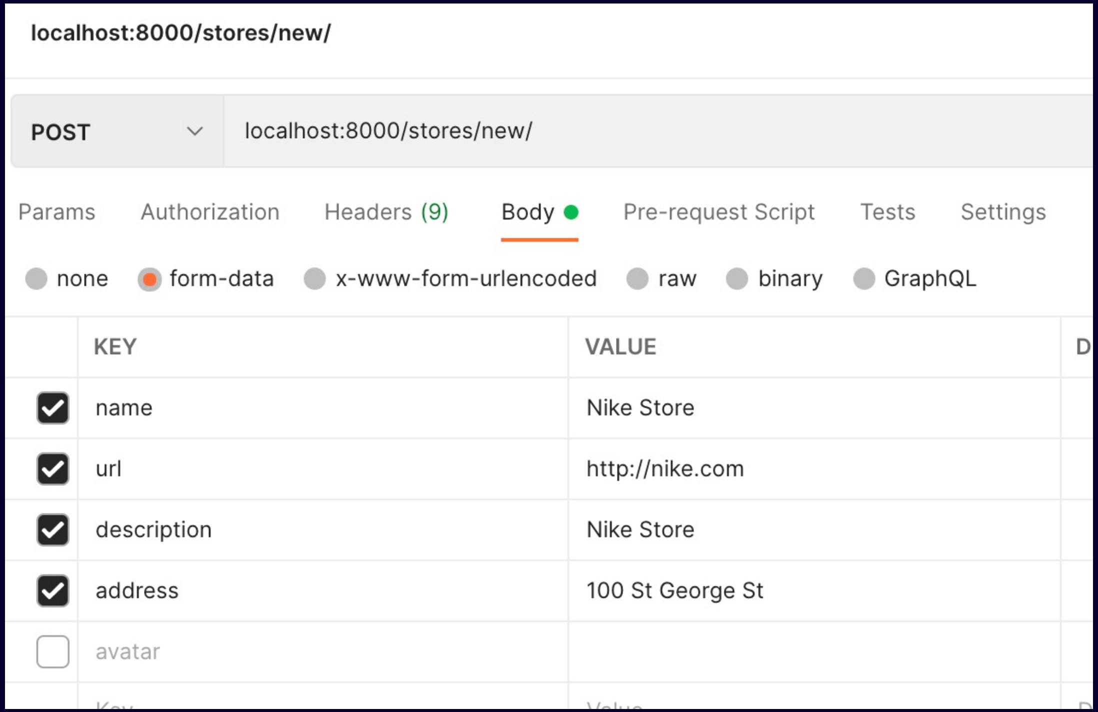
```
{
    "refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
        eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTY0NTE0MzI5OSwiaWF0IjoxNjQ1MDU2ODk5LCJqdGkiOiI2Nzg0ZjUyNT
        [illegible]",
    "access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
        eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjQ1MTQzMjk5LCJpYXQiOjE2NDUwNTY4OTksImp0aSI6ImU0OTkyMmQzMj
        RlMzRmZDFiYjlhMmE0YWZiOGEzMjI4IiwidXNlcl9pZCI6NX0.dprkzDWzrjWS6yzvpMrn7fXzACd-5ekzHMsxCXC6N3I"
}
```

Request to CreateAPIView with token



POST localhost:8000/stores/new/

	KEY	VALUE
✓	Authorization	Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI...
	Key	Value



localhost:8000/stores/new/

POST localhost:8000/stores/new/

none [✓ form-data] x-www-form-urlencoded raw binary GraphQL

	KEY	VALUE
✓	name	Nike Store
✓	url	http://nike.com
✓	description	Nike Store
✓	address	100 St George St
	avatar	

UpdateAPIView

- Implements both PUT and PATCH methods

- PUT does a full update, PATCH does a partial one

- `get_object` must be implemented

- The same serializer can be used
  - Read-only fields are discarded for deserialization
  - Method `update` can be overridden for special behavior

Permissions

- A set of permissions can be applied to APIViews
  Example: `IsAuthenticated`

- Specify permissions at the view
  `permission_classes = [IsAuthenticated]`

- Custom permission classes can be created as well
  Subclass `BasePermission` and implement `has_permission`

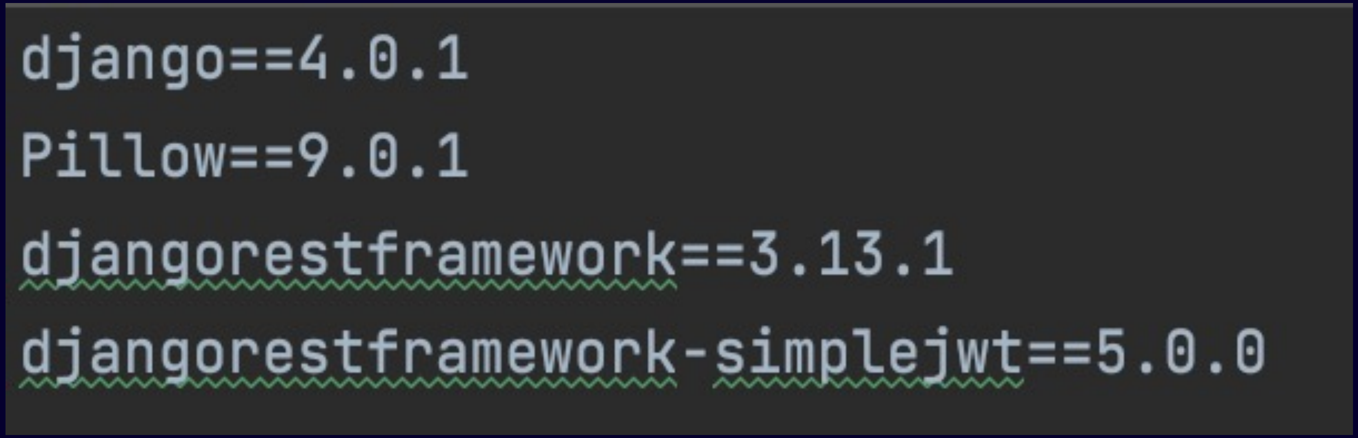
Requirements file

- Packages get out of hand very soon!
- If someone clones your code, they will be very confused about what to install
- It’s not only about the packages, but also their specific versions
  - Python projects are not backward-compatible

Requirements file

- Create a file named requirements.txt at project's root
- Once you `pip install` anything, add it (with its version) to that file
- To install packages from file:

```
pip install -r requirements.txt
```



```
django==4.0.1
Pillow==9.0.1
djangorestframework==3.13.1
djangorestframework-simplejwt==5.0.0
```

This session

- More on CRUD views
- Restful APIs
  VERY IMPORTANT: Project phase 2 is all about it!
- JSON, APIViews, serializers, permissions
- That whas is with back-end!

Next session

- Begins (or resumes) our front-end journey
- Intro to JavaScript
- DOM, finding elements, events, sessions
- Asynchronous requests (Ajax)

Final notes

- Next week is the reading week: No class!
- Lectures become dual (in-person with live-stream over Zoom)!
- Register your team for the phase 1 interview
- Assignment 2 due is on next Friday
- Use reading week to get started with project phase 2
  Design your database models first.
  Practice DRF. Phase 2 is all restful!