CSC309H1S

Programming on the Web

Winter 2023

Lecture 7: Django Forms and REST API

Instructor: Kuei (Jack) Sun

Department of Computer Science
University of Toronto

v1.02

Django Form

- https://docs.djangoproject.com/en/4.1/topics/forms/
- An abstraction for working with HTML forms
  - *Frontend*: renders form; converts Django fields to HTML input elements
  - *Backend*: sanitizes and validates form data
- `Form` class
  - Extremely similar to a Django model class

```
from django import forms
class NameForm(forms.Form):
    name = forms.CharField(label='Your name', max_length=100)
```

⬇

```
<label for="id_name">Your name:</label>
<input type="text" name="name" maxlength="100" required id="id_name">
```

Making Django Form

- Convention for large projects
  - Create a forms directory and put each form class in a separate file
    - Add __init__.py and *import each form class*
- `clean` method
  - Performs *validation* (sanitization has been done already)
  - Override to add custom logic

```
def clean(self):
    data = super().clean()
    user = authenticate(username=data['username'], password=data['password'])
    if user:
        data['user'] = user
        return data
    raise ValidationError({'username' : ‘Bad username or password'})
```

Model Form

- https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/
- Form that maps closely to Django model

```
class ArticleForm(forms.ModelForm):
    class Meta:
        model = Article
        fields = ['pub_date', 'headline', 'content', 'reporter']
```

- `Meta` inner class
  - Defines the associated model and the fields that appear in the form
- `save` method
  - Create or update the associated Model object

```
f = ArticleForm(request.POST)
article = f.save()
```

Using Django Form

- With a function-based view:

```
def get_name(request):
    if request.method == 'POST':
        form = NameForm(request.POST)
        if form.is_valid():
            # process form here
            return HttpResponseRedirect('/thanks/')
    else:
        form = NameForm()
    return render(request, 'name.html', {'form': form})
```

- With a class-based view:

```
class NameView(FormView):
    form_class = NameForm
    template_name = 'name.html'
    success_url = '/thanks/'
```

Form Widgets

- Forms can be passed into template and rendered
  - E.g., {{ form }}
  - Result would be based on the form renderer (can be customized)
- Some form fields can be rendered differently
  - E.g., a CharField can be rendered as text input, password input, textarea, etc.
  - Specify a widget to customize the rendering

```
class LoginForm(forms.Form):
    username = forms.CharField(max_length=150)
    password = forms.CharField(widget=forms.PasswordInput())
```

- Not recommended for large projects
  - In MVC pattern, view should be separate from controller

Form View

- One of Django’s generic editing view
- Similar to other CRUD views, except more customizable
- `form_valid` method
  - Called when form is valid, i.e., the POST request contains valid data
  - Where business logic should be placed

```
class LoginView(FormView):
    form_class = LoginForm
    template_name = 'accounts/login.html'
    success_url = reverse_lazy('accounts:admin')
    def form_valid(self, form):
        login_user(self.request, form.cleaned_data['user'])
        return super().form_valid(form)
```

- `form_invalid` method: override to custom handle invalid data

CreateView and UpdateView

- `CreateView` class
  - A subclass of `FormView` whose `form_class` is a `ModelForm`
- `UpdateView` class
  - A subclass of `CreateView` that implements the `get_object` method
- A default `form_valid` method is implemented that saves the object

```
class SignupView(CreateView):
    form_class = SignupForm
    template_name = 'accounts/signup.html'
    success_url = reverse_lazy('accounts:welcome')

    def form_valid(self, form):
        self.request.session['from'] = 'signup'
        return super().form_valid(form)
```

Automatically saves the model object, i.e., User

Authenticated Views

- Simplifies views where user must be logged in
- Function-based views:

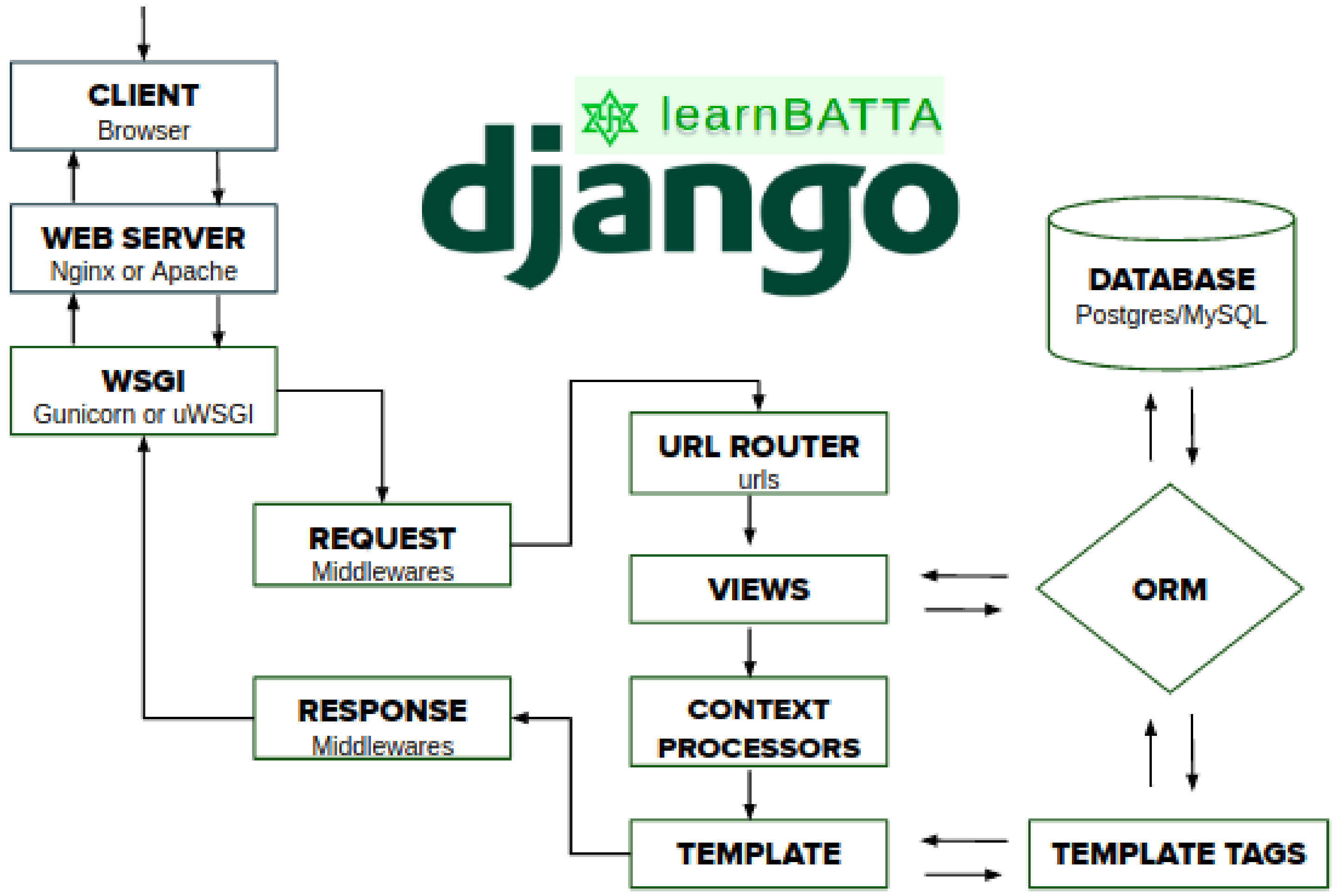
```
from django.contrib.auth.decorators import login_required
@login_required(login_url=reverse_lazy('accounts:login'))
def admin(request):
    return render(request, "accounts/admin.html", {})
```

- Class-based views:
  - Requires `login_url` to be specified for redirect

```
from django.contrib.auth.mixins import LoginRequiredMixin
class DeleteUserView(LoginRequiredMixin, DeleteView):
    model = User
    login_url = reverse_lazy('accounts:login')
    success_url = reverse_lazy('accounts:admin')
```

REST APIs

Current way of building Django website



request-response lifecycle in Django

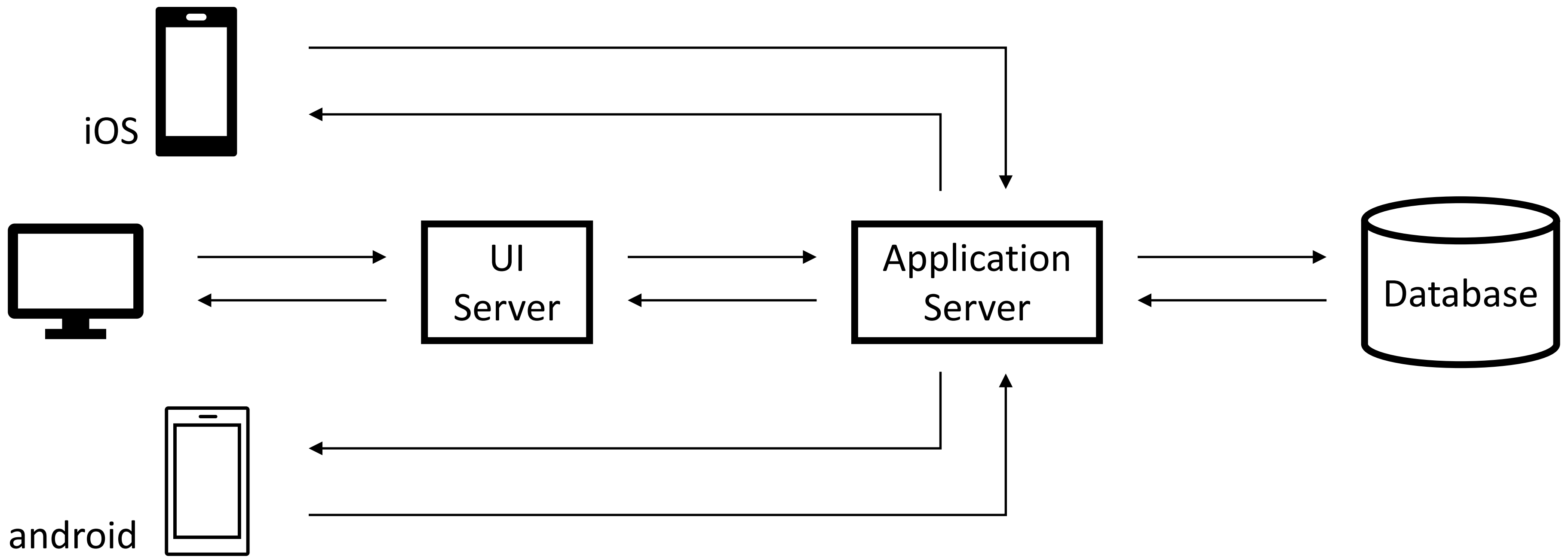
Source: https://learnbatta.com/blog/understanding-request-response-lifecycle-in-django-29/

Full Stack Framework

- Django is a full-stack framework
  - Libraries that do both backend and frontend work
- Server responsible for serving static files and handling business logic
- Design couples backend and frontend
  - Poor separation of duties
  - Can’t use a dedicated frontend framework like React
  - Restricts and/or complicates other types of *rendering pattern*
- Rendering pattern
  - The way HTML is rendered on the web
  - Django primarily supports server-side rendering

Separating Frontend and Backend

- Enables one backend and *multiple frontends*
  - e.g., web, android, iOS
- Improves modularity
  - Changes in frontend will not affect backend, and vice versa



Web API

- Different services and/or applications talk to each other
  - With a preestablished protocol
- API (application programming interface)
  - The way in which applications communicate with each other
- Web applications typically communicates via HTTP requests
- Backend views are responsible for data retrieval and manipulation
  - Should not care about how data is presented
    - e.g., should not handle templates or static files
      - i.e., does not need to work with HTML or CSS
- How should frontend and backend communicate then?

JavaScript Object Notation

- Popular standard for backend responses
- Derived from JavaScript syntax for defining objects
  - Simplifies use in a browser, which supports JavaScript natively
- Advantages
  - Easy to read, easy to use, and fast
  - Many programming languages have built-in parser and support

- Example:

```json
[
  {
    "_id": "63ea43564bfe5fbf662a2e76",
    "index": 0,
    "guid": null,
    "isActive": false,
    "balance": "$3,863.93",
    "picture": "http://placehold.it/img",
    "age": 20,
    "name": "Duffy Sanchez",
    "friends": [
      { "id": 0, "name": "Rosie Crell" },
      { "id": 1, "name": "Eaton Mars" },
    ],
    "favoriteFruit": "strawberry"
  }
]
```

JSON Format

- Primitives types
  - number, string, boolean and null
- Array
  - Ordered collection of elements
- Object
  - Key-value pairs
  - Key must always be a string
- Array elements and object values can be of *any* type
  - Primitive or aggregate

- Example:

```
[
  {
    "_id": "63ea43564bfe5fbf662a2e76",
    "index": 0,
    "guid": null,
    "isActive": false,
    "balance": "$3,863.93",
    "picture": "http://placehold.it/img",
    "age": 20,
    "name": "Duffy Sanchez",
    "friends": [
      { "id": 0, "name": "Rosie Crell" },
      { "id": 1, "name": "Eaton Mars" },
    ],
    "favoriteFruit": "strawberry"
  }
]
```

JSON Exercise

- Given the following tables where each store has an owner, *serialize* the User with username jack. *Nest* all related data.
- Store

id	name	url	email	is_active	owner_id
1	Apple	https://www.apple.com	apple@test.com	1	1
2	Adidas	https://www.adidas.com	adidas@test.com	1	2
3	Nike	https://www.nike.com	nike@test.com	1	1
4	Sobeys	https://www.sobeys.com	sobeys@test.com	1	null

- User

id	username	first_name	last_name	email	last_login
1	jack	Jack	Smith	jack@test.com	2023-02-10 07:23:53.568000

Web APIs

- REST (Representation State Transfer)
  - A particular architectural style with a set of constraints and principles
  - Goal is to create a scalable, maintainable, and flexible system
  1. Uses HTTP verbs to make requests, e.g., GET, PUT, POST, etc.
     - Resource should be identified through URIs
  2. Requires stateless client-server communication
  3. Responses should be clearly labeled as cacheable or non-cacheable
  4. Client should only interact with the API and not server directly
- SOAP (Simple Object Access Protocol)
  - XML-based protocol with standardized format for data transfer
  - Less popular now, due to advent of REST

Django REST Framework (DRF)

Django REST framework

- Helps with writing RESTful APIs
- Provides JSON parser, CRUD views, permissions, and serializers
- Only uses Django's backend
  - Models and URLs are unchanged
  - Views are subclasses of DRF views
- Installation
  - `pip3 install djangorestframework`
  - Add 'rest_framework' to INSTALLED_APPS in settings.py and this:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny'
    ]
}
```

No authentication required. Do not use.

REST Views

- Same idea, but returns a REST Response class
  - Takes a list or a dictionary, and converts it to an HTTP JSON response

- Function-based view

```
from rest_framework.decorators \
     import api_view

@api_view(['GET'])
def stores_list(request):
    stores = Store.objects.filter( \
             is_active=True)
    return Response([
    {
        'name' : store.name,
        'url' : store.url,
    }
    for store in stores ])
```

- Class-based view

```
from rest_framework.response \
     import Response
from rest_framework.views import APIView

class StoresManage(APIView):
    def get(self, request):
        stores = Store.objects.all()
        return Response([
        {
            'name' : store.name,
            'url' : store.url,
        }
        for store in stores ])
```

Model Serializer

- Model instances need to be serialized and deserialized for client
- Object represented in format that can be transferred and reconstructed
- DRF provides JSON serializer
  - Very similar in flavor as Django Model Form
  - Plain serializer (not mapped to a model) also available
- Create a serializer.py or a serializers directory in the app

```python
from rest_framework.serializers import ModelSerializer

class StoreSerializer(ModelSerializer):
    class Meta:
        model = Store
        fields = ['name', 'url', 'email', 'is_active']
```

REST CRUD views

- Same idea, but requires a model serializer instead
- CreateAPIView
  - Overrides create method (returns 201 Created on success, accepts HTTP POST)
- RetrieveAPIView and ListAPIVIew
  - Overrides retrieve method (returns 200 OK on success, accepts HTTP GET)
- UpdateAPIView
  - Overrides update method (returns 200 OK on success)
    - Provides HTTP PUT and PATCH method handlers
- DestroyAPIView
  - Overrides destroy method (returns 204 No Content on success)
    - Provides HTTP DELETE method handler

More about CRUD views

- ListAPIView
  - Requires `queryset` attribute or `get_queryset` method
- RetrieveAPIView, UpdateAPIView, DeleteAPIView
  - Requires `get_object` method
- CreateAPIView
  - Does not require any addition method or attribute
- You can mix multiple views in one class, i.e., multiple inheritance
  - Works as long as each view uses a different HTTP method
- Can use same serializer across different views
  - In some cases, you may want to create separate serializers

Example

- Retrieve Store View

```
from django.shortcuts import get_object_or_404
from rest_framework.generics import RetrieveAPIView

class StoresRetrieve(RetrieveAPIView):
    serializer_class = StoreSerializer
    def get_object(self):
        return get_object_or_404(Store, id=self.kwargs['pk'])
```

Django shortcut that returns 404 NOT FOUND if the object is not found

- Testing
    - Postman or DRF’s built-in browsable APIs in development mode

Serialization Fields

- Fields have similar options to Django's model field
  - Exceptions: null → `allow_null`, blank → `allow_blank`
  - `read_only`: makes a field non-writeable
- Field validations are done automatically
- Foreign Key
  - By default, serializes to id of referenced object
- Custom fields
  - Can create new fields or override existing fields

```
class StoreSerializer(ModelSerializer):
    owner_username = CharField(read_only=True,
        source='owner.username', allow_null=True)
    ...
```

Token-Based Authentication

REST Authentication

- DRF’s browsable API works with session auth
  - However, REST APIs must be stateless!
- REST APIs uses token-based authentication
- JWT (JSON Web Token) packages
  - The other package is deprecated; therefore, we will use simplejwt
    - https://django-rest-framework-simplejwt.readthedocs.io/en/latest/getting_started.html
- Installation
  - `pip3 install djangorestframework-simplejwt`

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

Setting up simplejwt

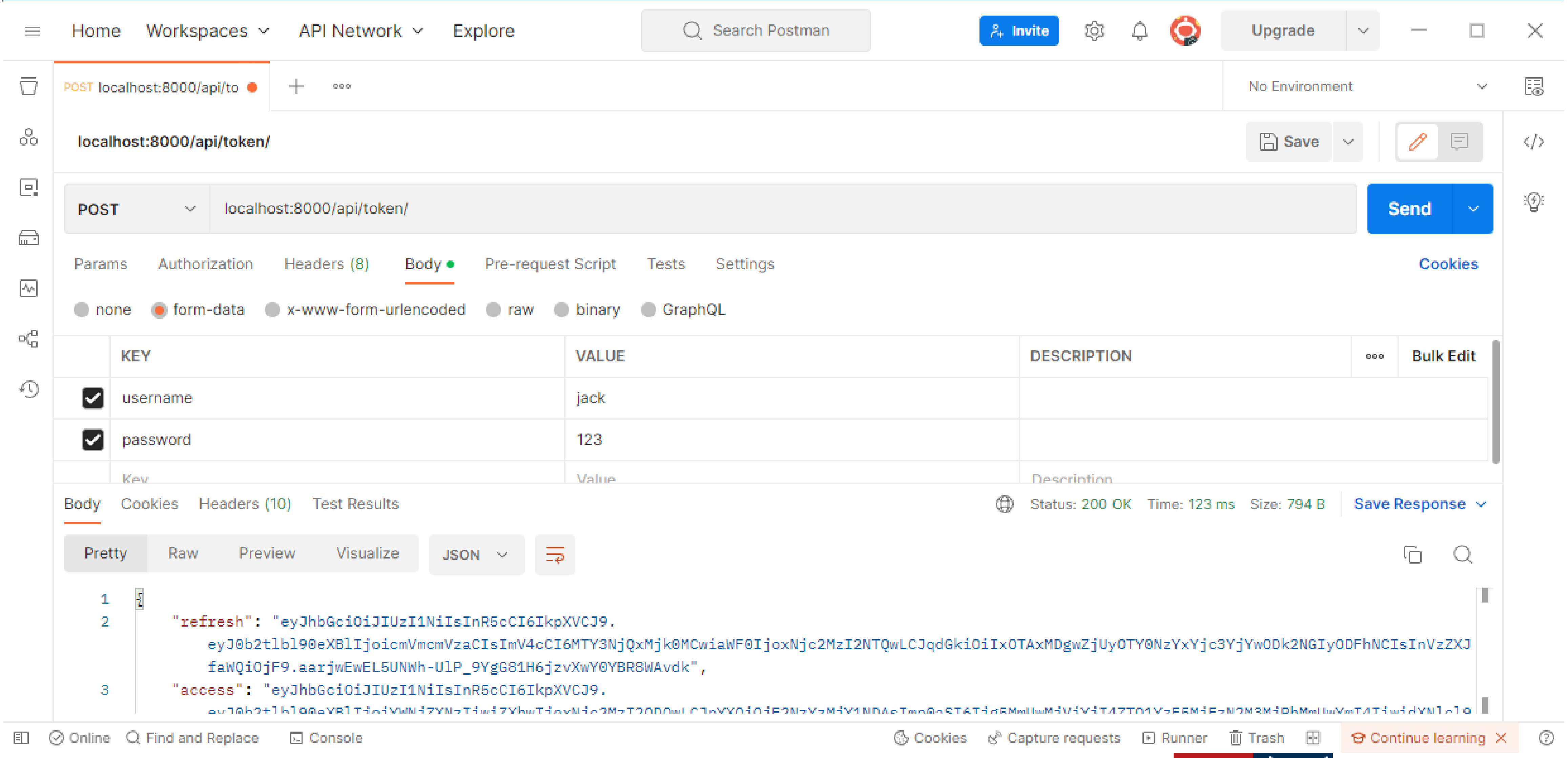
- Create login view (provided by simplejwt)

```
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView

urlpatterns = [
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
]
```

- Token is short-lived
  - Five minutes by default
  - Can be changed to other durations
    - https://django-rest-framework-simplejwt.readthedocs.io/en/latest/settings.html
  - A refresh token can be used to extend its duration

Obtaining a Token



POST localhost:8000/api/token/

Body: form-data

KEY	VALUE	DESCRIPTION
✓ username	jack	
✓ password	123	

Status: 200 OK Time: 123 ms Size: 794 B

```
{
    "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTY3NjQxMjk0MCwiaWF0IjoxNjc2MzI2NTQwLCJqdGkiOiIxOTAxMDgwZjUyOTY0NzYxYjc3YjYwODk2NGIyODFhNCIsInVzZXJ
        faWQiOjF9.aarjwEwEL5UNWh-UlP_9YgG81H6jzvXwY0YBR8WAvdk",
    "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
```

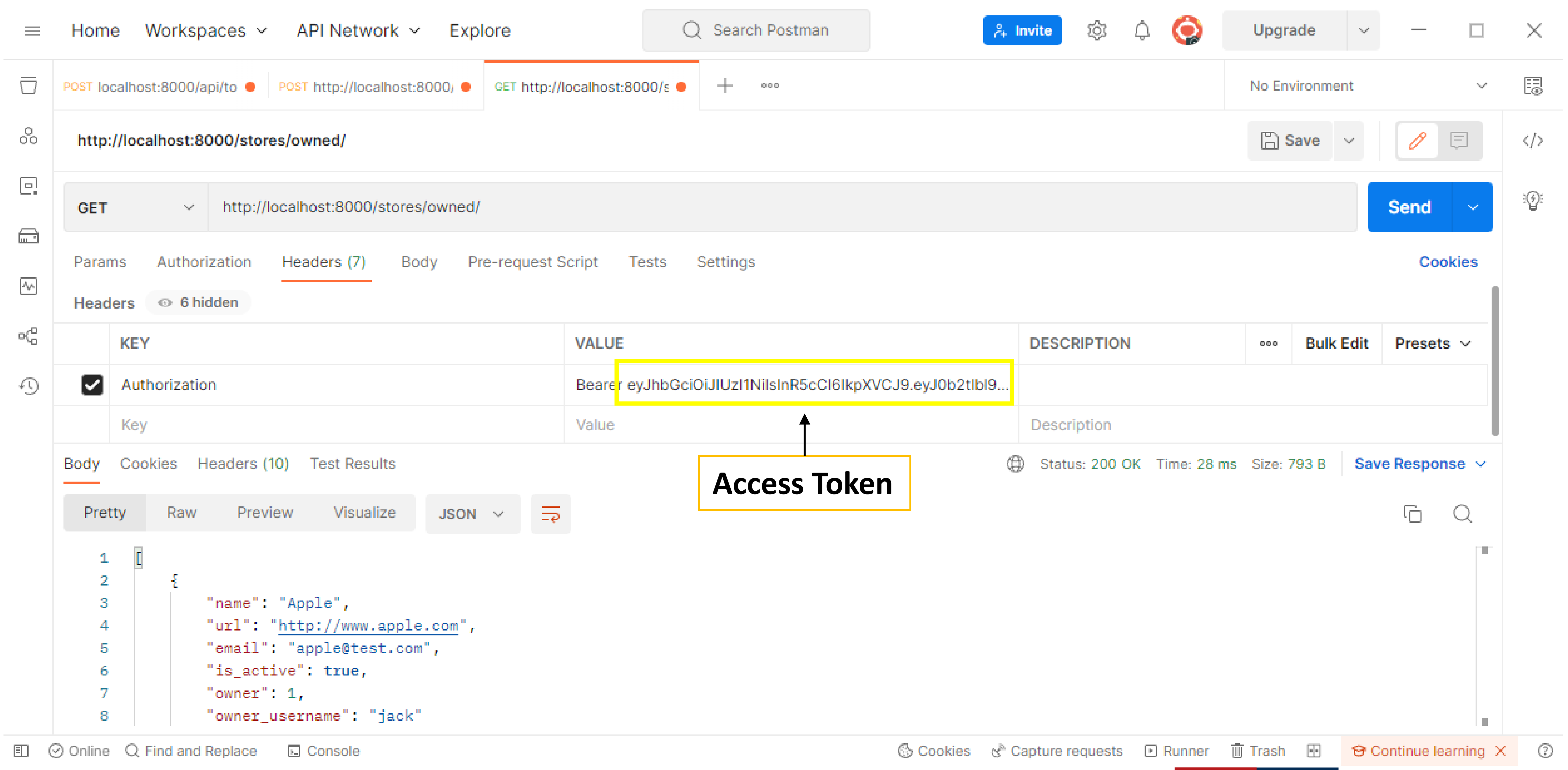
REST Permissions

- A set of permissions can be applied to APIViews
  - E.g.: `IsAuthenticated`
    - This requires the user to be logged in, e.g., via token
- Can specify a list of permissions for a view

```
from rest_framework.permissions import IsAuthenticated
class StoresOwned(ListAPIView):
    permission_classes = [IsAuthenticated]
    serializer_class = StoreSerializer
    def get_queryset(self):
        return Store.objects.filter(owner=self.request.user)
```

- Custom permissions can be created as well
  - Subclass `BasePermission` and implement `has_permission` method

Using a Token



http://localhost:8000/stores/owned/

GET http://localhost:8000/stores/owned/

Headers

	KEY	VALUE	DESCRIPTION
✓	Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl9...	
	Key	Value	Description

Body Cookies Headers (10) Test Results — Status: 200 OK Time: 28 ms Size: 793 B

```
[
    {
        "name": "Apple",
        "url": "http://www.apple.com",
        "email": "apple@test.com",
        "is_active": true,
        "owner": 1,
        "owner_username": "jack"
```