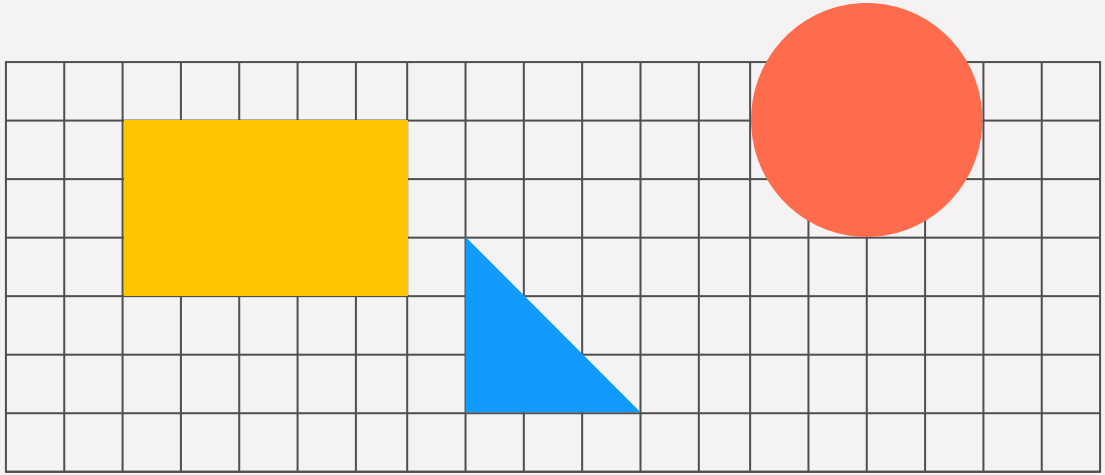
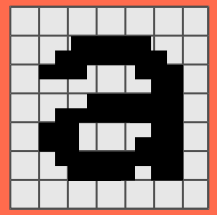
Beyond Requests and Responses!

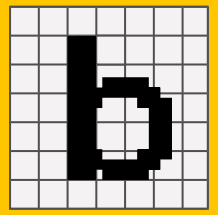
Ali Abrishami



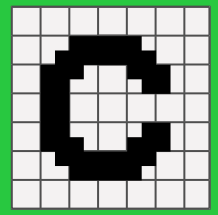
In this Lecture You will...



Learn Advanced Concepts of DRF



Design & Develop Secure APIs



Make use of DRFs lesser known utilities

REMINDER: REST

- REST is an architectural style, not a framework, library, or protocol you can install.
- Resources are the core concept; URLs name things, not actions.
- HTTP methods have meaning: GET reads, POST creates, PUT replaces, PATCH modifies, DELETE removes.
- Statelessness is non-negotiable: each request must contain all required context.
- HTTP status codes are part of the API contract, not decorative numbers.

REMINDER: Access Control

- **Authentication** answers who the requester is; it establishes identity, nothing more
- **Authorization** (permissions) answers what an authenticated actor is allowed to do
- Object-level access control enforces rules per resource instance, not just per endpoint
- Access policies must encode business rules, not mirror database structure
- Access control is cross-cutting: it must be consistent across views, serializers, and actions

Permissions

- Permissions define what an authenticated principal is allowed to do, not who they are.
- Coarse-grained permissions protect endpoints; fine-grained permissions protect individual resources.
- Permission checks must reflect business rules, not URL structures or HTTP methods.
- Read and write access are distinct concerns and should be modeled separately.
- Permissions are enforcement points, not documentation. Failure **must** be **explicit** and **consistent.**

Permissions in DRF

- Permissions are enforced at the **view layer**, before request handling completes.
- DRF evaluates permissions per request and can re-evaluate them per object.
- Multiple permission classes are combined with **logical AND** by default.
- Object-level permissions require explicit invocation during data access.
- Custom permission classes are the proper place for business authorization rules.

Common Perm. Strategies

- Public-read, authenticated-write for shared resources.
- Owner-based permissions for user-generated content.
- Role-based permissions layered on top of global checks.
- Read-only access for non-privileged actors.
- State-aware permissions tied to lifecycle stages of a resource.

Code-level Inspection

- Permission classes inherit from `BasePermission` and implement `has_permission` and optionally `has_object_permission`
- `has_permission` runs before queryset evaluation; `has_object_permission` runs per instance.
- Permissions are declared explicitly on views via `permission_classes`, not inferred.
- Multiple permission classes are evaluated sequentially and short-circuit on failure.
- Object-level permissions must be enforced manually in generic views (`check_object_permissions`).

Custom Perm. Classes!

todolist/tasks/views.py

```python
from rest_framework.permissions import BasePermission

class IsSuperUser(BasePermission):
    def has_permission(self, request, view):
        return (
            request.user
            and request.user.is_authenticated
            and request.user.is_superuser
        )
```

Update The View

todolist/tasks/views.py

```
class TaskListCreateAPIView(ListCreateAPIView):
    queryset = Task.objects.all().order_by("-creation_date")
    serializer_class = TaskSerializer
    pagination_class = TaskPagination
    permission_classes = [IsSuperUser]
```

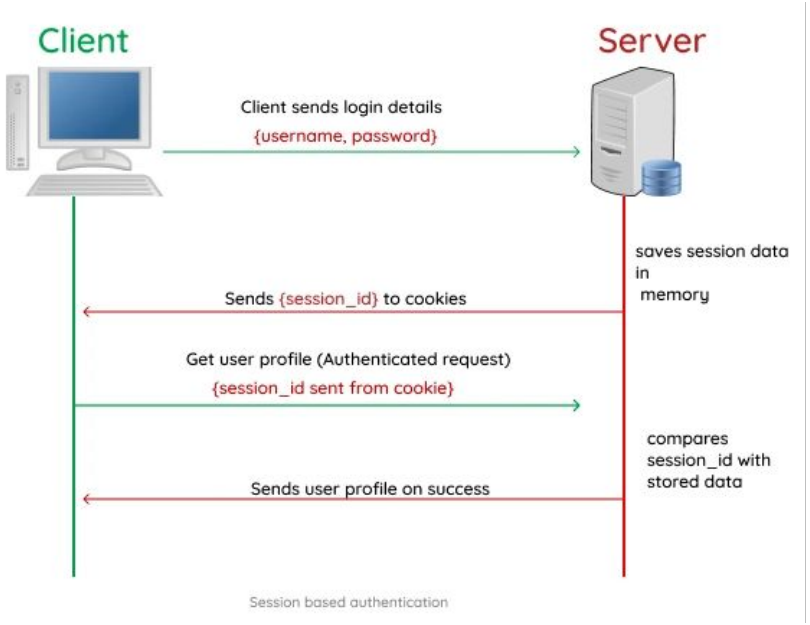
Protecting Endpoints

todolist/tasks/views.py

```python
class TaskListCreateAPIView(ListCreateAPIView):
    queryset = Task.objects.all().order_by("-creation_date")
    serializer_class = TaskSerializer
    pagination_class = TaskPagination

    def get_permissions(self):
        if self.request.method == "POST":
            return [IsAuthenticated()]
        return [AllowAny()]
```

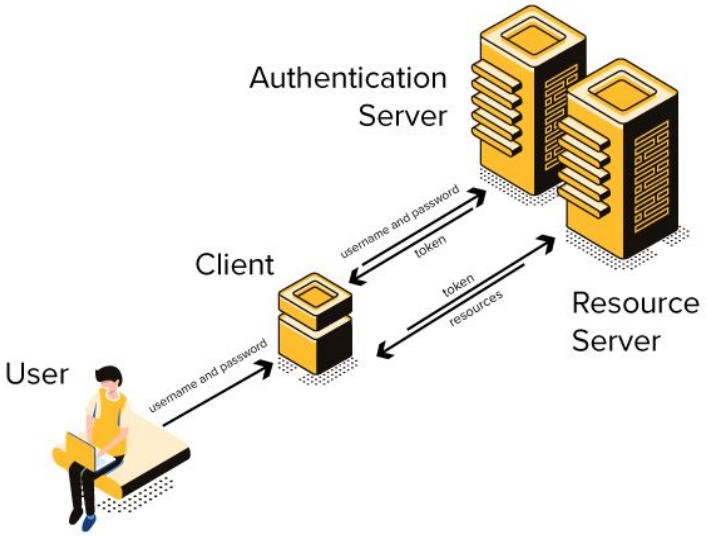
Session Based Auth

- Uses Django's built-in auth system and server-side session storage
- Authentication state is managed via cookies and middleware
- DRF permissions rely directly on the resolved `request.user`
- Simple integration with Django admin and traditional web views.
- Limited suitability for pure APIs and distributed backends.



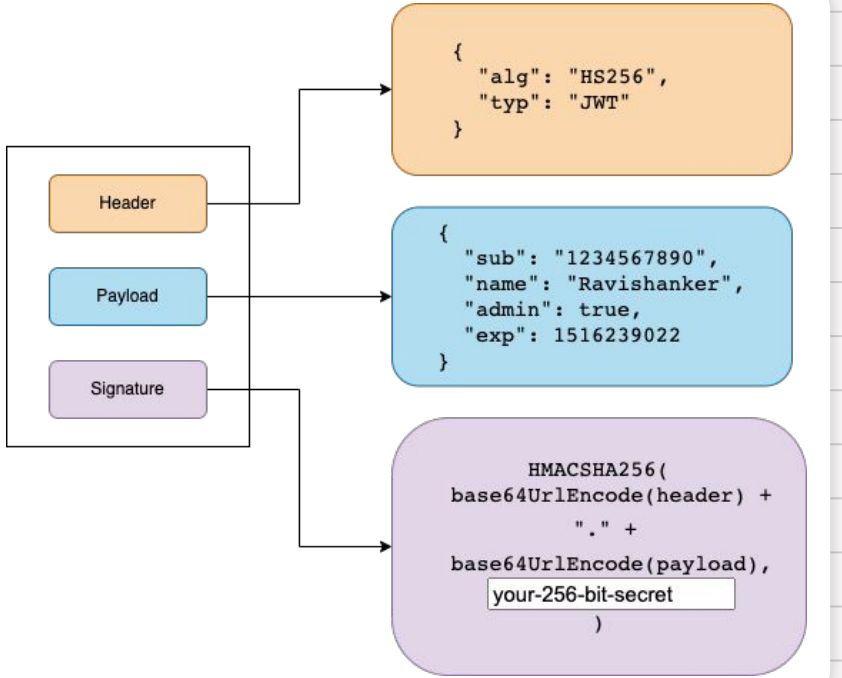
Token Based Auth

- Client authenticates once and receives an opaque token.
- Token is sent with every request, typically via the `Authorization` header.
- Server validates the token and resolves a user for the request.
- DRF permissions operate on the resolved user exactly as with sessions.
- Token storage, expiry, and revocation are explicit design concerns.



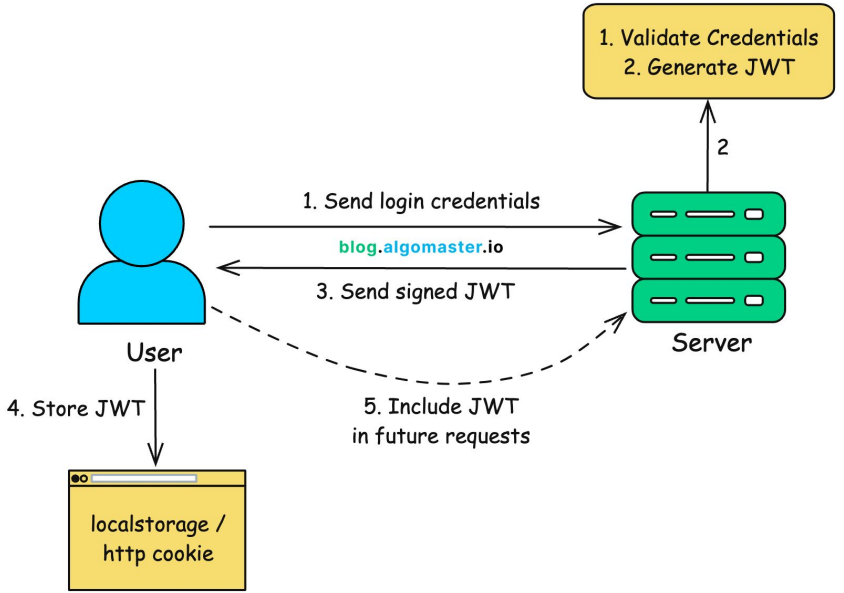
JWT!

- A compact, URL-safe token used to securely transmit claims between parties
- Consists of three parts: header, payload (claims), and signature
- Digitally signed to ensure integrity, not confidentiality by default
- Self-contained: the server can verify it **without** storing session state
- Represents asserted identity and context, not granted permissions



JWT Properties

- **Access token**: short-lived proof of authentication (sent on every request)
- **Refresh token**: longer-lived token to mint new access tokens (stored more carefully)
- JWT is **stateless** verification (signature) but your app still needs stateful policy (revocation/rotation decisions)
- Your threat model shifts: token theft becomes your “password leak” equivalent



Setup JWT in DRF

```
floaterm(1/1)
 mani  pip install djangorestframework-simplejwt                    drf
Collecting djangorestframework-simplejwt
  Downloading djangorestframework_simplejwt-5.5.1-py3-none-any.whl.metadata (4.6 kB)
Requirement already satisfied: django⩾4.2 in ./venv/lib/python3.13/site-packages (from djangorestfram
ework-simplejwt) (5.2.1)
Requirement already satisfied: djangorestframework⩾3.14 in ./venv/lib/python3.13/site-packages (from
djangorestframework-simplejwt) (3.14.0)
Collecting pyjwt⩾1.7.1 (from djangorestframework-simplejwt)
  Downloading PyJWT-2.10.1-py3-none-any.whl.metadata (4.0 kB)
```

Projects/University/web-ta/sampleprojects/drf/todolist/todolist/todolist/settings.py

```python
REST_FRAMEWORK = {
    "DEFAULT_SCHEMA_CLASS": "drf_spectacular.openapi.AutoSchema",
    "DEFAULT_AUTHENTICATION_CLASSES": (
        "rest_framework_simplejwt.authentication.JWTAuthentication",
    ),
    "DEFAULT_PERMISSION_CLASSES": ("rest_framework.permissions.IsAuthenticated",),
}
```

Bind Views to URLs

todolist/authservice/urls.py

```python
from django.urls import path
from rest_framework_simplejwt.views import (
    TokenObtainPairView,
    TokenRefreshView,
    TokenVerifyView,
)

urlpatterns = [
    path("token/", TokenObtainPairView.as_view(), name="token_obtain_pair"),
    path("token/refresh/", TokenRefreshView.as_view(), name="token_refresh"),
    path("token/verify/", TokenVerifyView.as_view(), name="token_verify"),
]
```

Tweak JWT Settings

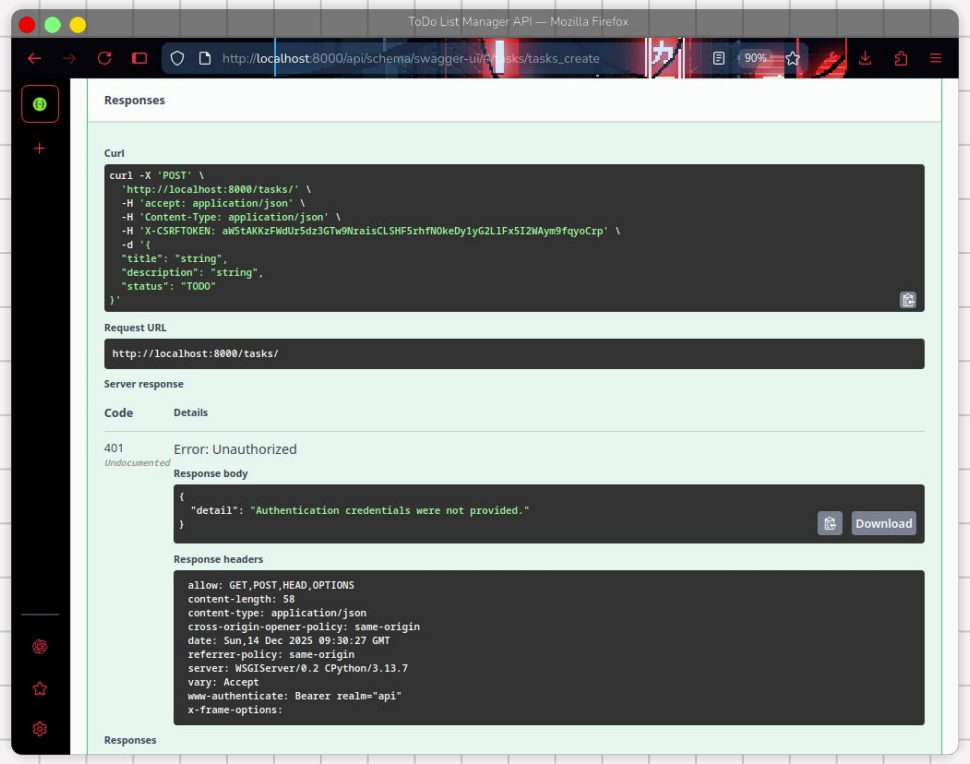
todolist/todolist/settings.py

```python
SIMPLE_JWT = {
    "ACCESS_TOKEN_LIFETIME": timedelta(minutes=10),
    "REFRESH_TOKEN_LIFETIME": timedelta(days=7),

    "ROTATE_REFRESH_TOKENS": True,

    "AUTH_HEADER_TYPES": ("Bearer",),
}
```

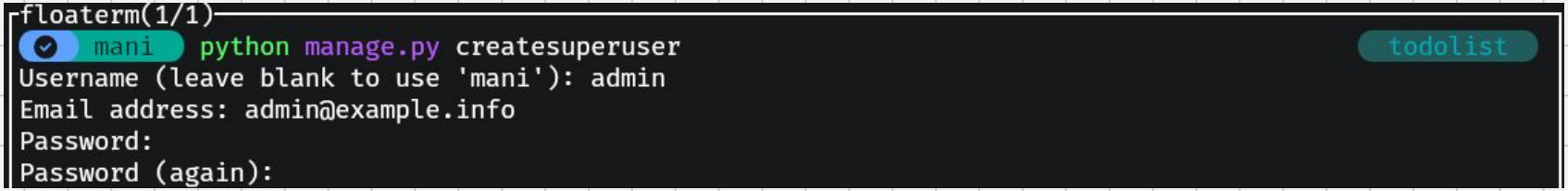
Current Auth Flow

- Client calls `/auth/token/` with credentials and gets access and refresh tokens.
- Client stores: access (short-lived, used on API calls) and refresh (long-lived, used only to mint new access tokens) tokens
- Each request contains a `Authorization: Bearer <access>` header
- On 401 due to expiry, call `/auth/token/refresh/` with refresh and get new access (and possibly new refresh)

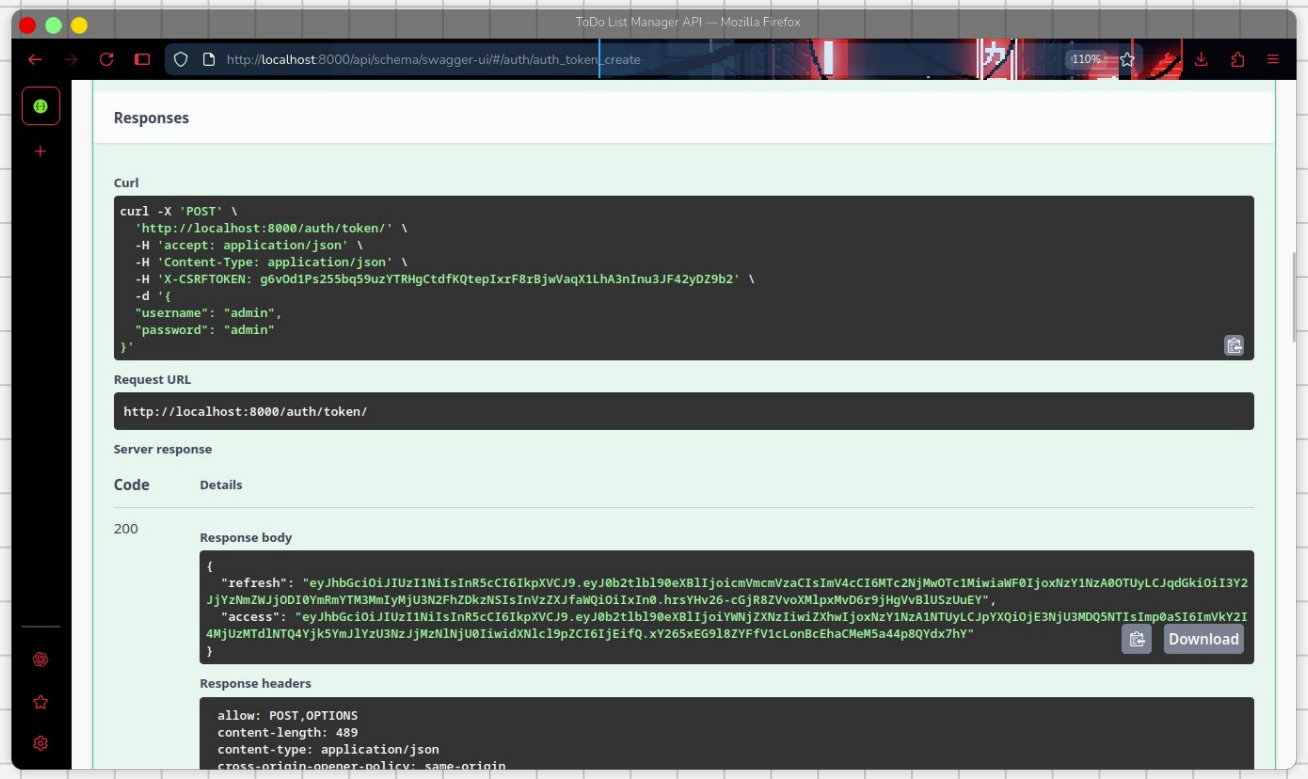
IT WORKS!


ToDo List Manager API — Mozilla Firefox
http://localhost:8000/api/schema/swagger-ui/#/tasks/tasks_create
90%
Responses
Curl
curl -X 'POST' \
'http://localhost:8000/tasks/' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-H 'X-CSRFTOKEN: aW5tAKKzFWdUz5dz3GTw9NraisCLSHF5rhfNOkeDy1yG2L1Fx5I2WAym9fqyoCrp' \
-d '{
"title": "string",
"description": "string",
"status": "TODO"
}'
Request URL
http://localhost:8000/tasks/
Server response
Code
Details
401
Undocumented
Error: Unauthorized
Response body
{
"detail": "Authentication credentials were not provided."
}
Download
Response headers
allow: GET,POST,HEAD,OPTIONS
content-length: 58
content-type: application/json
cross-origin-opener-policy: same-origin
date: Sun,14 Dec 2025 09:30:27 GMT
referrer-policy: same-origin
server: WSGIServer/0.2 CPython/3.13.7
vary: Accept
www-authenticate: Bearer realm="api"
x-frame-options:
Responses

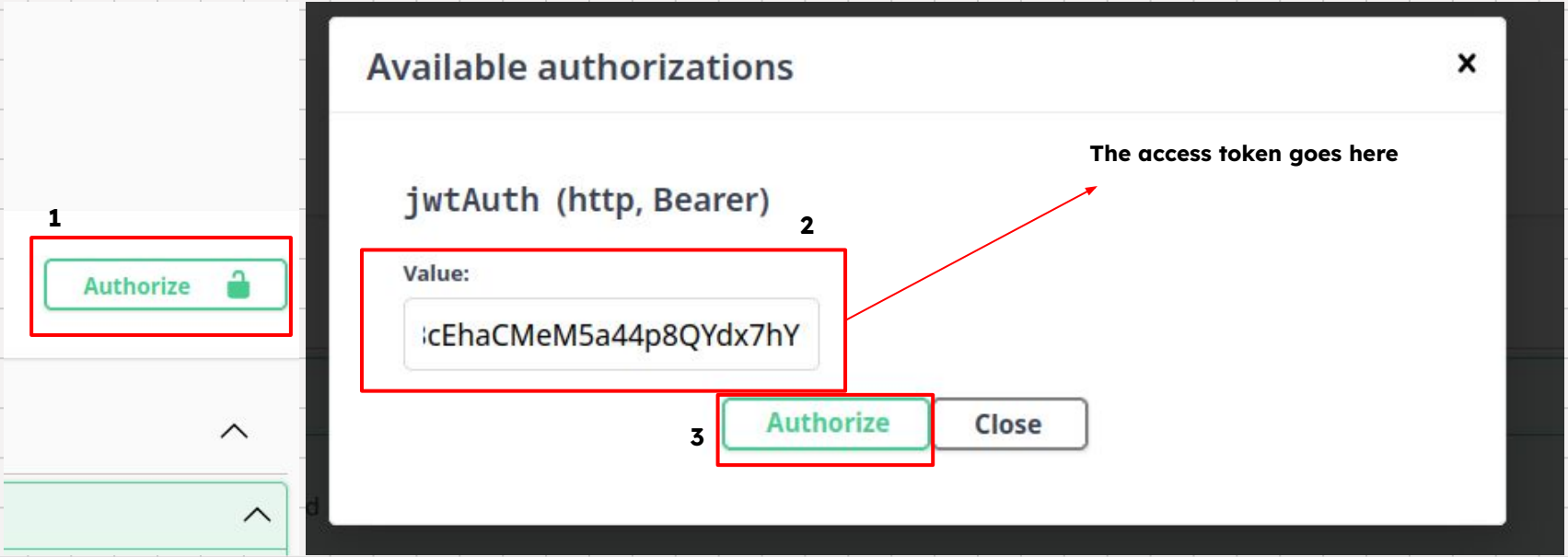
Create a Superuser



Log in


ToDo List Manager API — Mozilla Firefox
http://localhost:8000/api/schema/swagger-ui/#/auth/auth_token_create
110%
Responses
Curl
curl -X 'POST' \
'http://localhost:8000/auth/token/' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-H 'X-CSRFTOKEN: g6vOd1Ps255bq59uzYTRHgCtdfKQtepIxrF8rBjwVaqX1LhA3nInu3JF42yDZ9b2' \
-d '{
"username": "admin",
"password": "admin"
}'
Request URL
http://localhost:8000/auth/token/
Server response
Code
Details
200
Response body
{
"refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTc2NjMwOTc1MiwiaWF0IjoxNzY1NzA0OTUyLCJqdGkiOiI3Y2
JjYzNmZWJjODI0YmRmYTM3MmIyMjU3N2FhZDkzNSIsInVzZXJfaWQiOiIxIn0.hrsYHv26-cGjR8ZVvoXMlpxMvD6r9jHgVvBlUSzUuEY",
"access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzY1NzA1NTUyLCJpYXQiOjE3NjU3MDQ5NTIsImp0aSI6ImVkY2I
4MjUzMTdlNTQ4Yjk5YmJlYzU3NzJjMzNlNjU0IiwidXNlcl9pZCI6IjEifQ.xY265xEG9l8ZYFfV1cLonBcEhaCMeM5a44p8QYdx7hY"
}
Download
Response headers
allow: POST,OPTIONS
content-length: 489
content-type: application/json
cross-origin-opener-policy: same-origin

Pass the Token to Swagger



No More 401 or 403!

200

Response body


```
{
  "count": 4,
  "next": null,
  "previous": null,
  "results": [
    {
      "id": 4,
      "title": "Something",
      "description": "",
      "status": "TODO",
      "creation_date": "2025-12-12T18:54:08.394375Z"
    },
    {
      "id": 3,
      "title": "Anothersomething",
      "description": "",
      "status": "TODO",
      "creation_date": "2025-12-12T18:54:04.364344Z"
    },
    {
      "id": 2,
      "title": "Another",
      "description": "",
      "status": "TODO",
```


Download

Custom User Models

- Django allows replacing the default user model once, at project start
- A custom user model defines identity, not permissions or roles
- Typically used to change the login field (e.g. `username` to `email`) or add core attributes
- Implemented by subclassing `AbstractUser` or `AbstractBaseUser`
- Must be referenced via `AUTH_USER_MODEL` everywhere to avoid tight coupling. And of course, it must be specified in `settings.py`

Class Activity Time!

What `manage.py` commands do you think become unusable after you change the default user model? What can you do to fix such issues?

Hint for the second part: look up custom `manage.py` commands!

RBAC!

- RBAC (Role-Based Access Control) restricts system access based on predefined user roles
- Roles represent **responsibilities**, not individuals (e.g., admin, staff, user)
- Users inherit permissions through roles, avoiding per-user access rules
- Authorization is separated from authentication, improving clarity and security
- RBAC scales well for large systems with many users and endpoints

Method I: Django `group`

- Use Django's built-in Group model to represent roles
- Assign Permissions to groups, not directly to users
- Users inherit all permissions from their assigned groups
- Enforce access via DRF permission classes and `user.has_perm()`
- Fast to implement, stable, and well-integrated with Django Admin

Method II: Custom Roles

- Define an explicit `Role` model with custom fields and relationships
- Associate users with roles via `ForeignKey` or `ManyToMany` relations
- Encode permissions as enums, flags, or linked permission tables
- Enforce rules using custom DRF permission classes
- More flexible, but increases complexity and maintenance cost

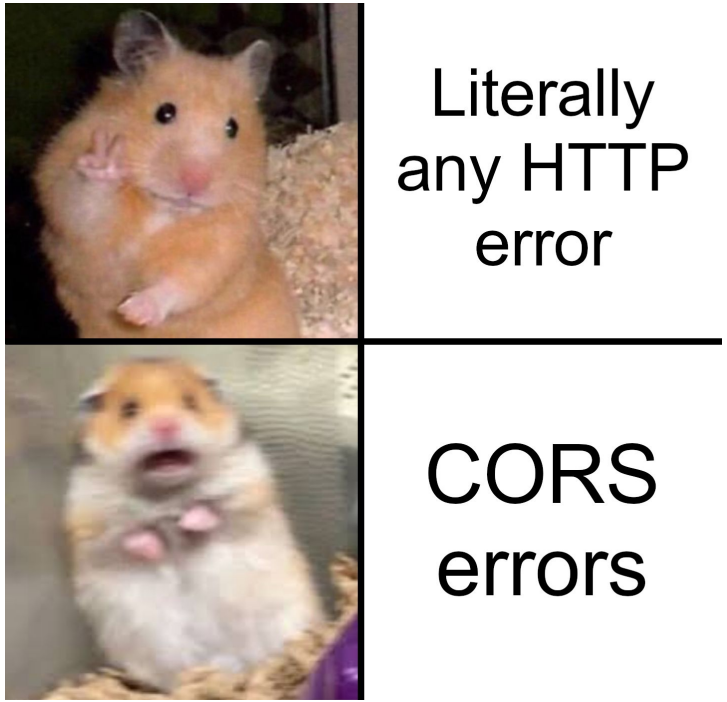
Class Activity Time!

What are the pros and cons of each one of the methods?

How do you handle role updates when using JWTs?

CORS!

- CORS is a browser security mechanism, not an API security feature
- It controls which origins are allowed to make cross-origin requests
- Enforced by browsers, completely ignored by non-browser clients
- Designed to protect users, not servers
- Misunderstanding CORS leads to false confidence in API security

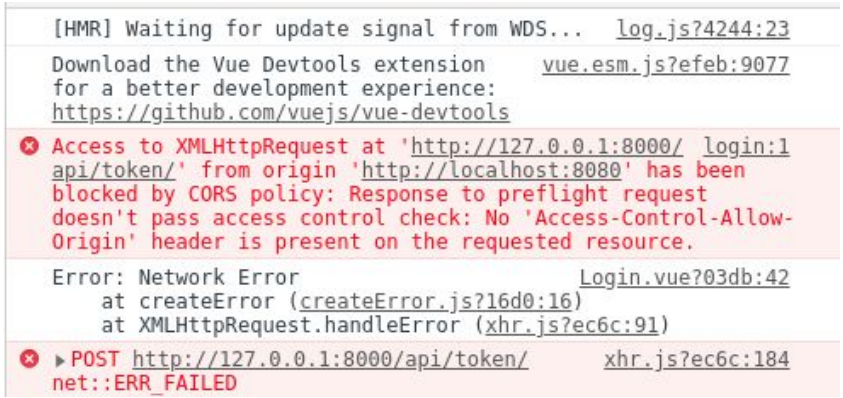


CORS Mechanism

- Browsers enforce the Same-Origin Policy by default
- CORS allows controlled relaxation of that policy
- Browser sends Origin; server replies with permission headers
- “Unsafe” requests trigger a preflight (OPTIONS) check
- Blocking happens in the browser, not on the server

Handle CORS in Django

- Django does not handle CORS by default
- CORS must be implemented at the middleware level
- The de-facto standard is `django-cors-headers`
- CORS applies to **all** responses, including errors and 401s
- Misconfiguration is silent and common

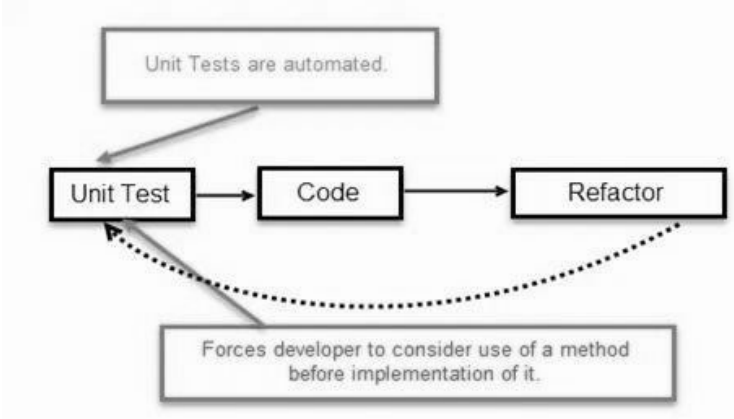


And the Pitfalls

- Placing CORS middleware after authentication or common middleware
- Using `CORS_ALLOW_ALL_ORIGINS = True` in production
- Forgetting credentialed requests need explicit origin allowance
- Assuming CORS replaces authentication or permissions

Testing Your DRF APIs

- Test API behavior, not internal implementation details
- Test Authentication and permission boundaries explicitly
- Test Correct status codes and error responses
- Test Serialization and validation rules at the API boundary
- Test Side effects: database state and business invariants



How DRF Tests Work

- Use DRF’s API client to simulate real HTTP requests
- Test authenticated vs unauthenticated access deliberately
- Assert permissions, not just successful responses
- Keep tests isolated, deterministic, and fast
- Treat tests as part of the API contract

Read more here

Sample DRF Test

todolist/tasks/tests.py

```python
User = get_user_model()


class TaskListCreateAPITests(APITestCase):

    def test_superuser_can_create_task(self):
        superuser = User.objects.create_superuser(username="admin", password="admin")
        self.client.force_authenticate(user=superuser)

        url = reverse("task-list-create")
        data = {"title": "Test Task", "description": "Created by superuser"}

        response = self.client.post(url, data)

        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
```

And... Action!

```
floaterm(1/1)
 mani  python manage.py test                                    todolist
Found 1 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.391s

OK
Destroying test database for alias 'default'...
```

Class Activity Time!

What does `reverse` do?

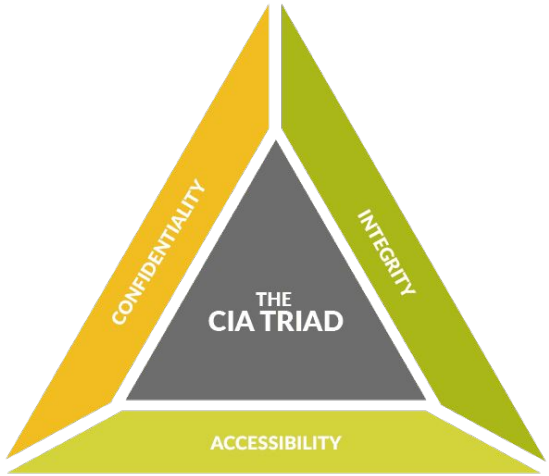
Where do you use `force_authenticate`?

Name some more assertion methods in tests.

READ MORE!

Security and CIA

- **Confidentiality** ensures data is accessible only to authorized entities
- **Integrity** guarantees data is accurate, unaltered, and trustworthy over time
- **Availability** ensures systems and data remain accessible when needed
- Security mechanisms always trade off between CIA properties.
- Real security design is risk-based. Protect what **matters most**, not everything equally.

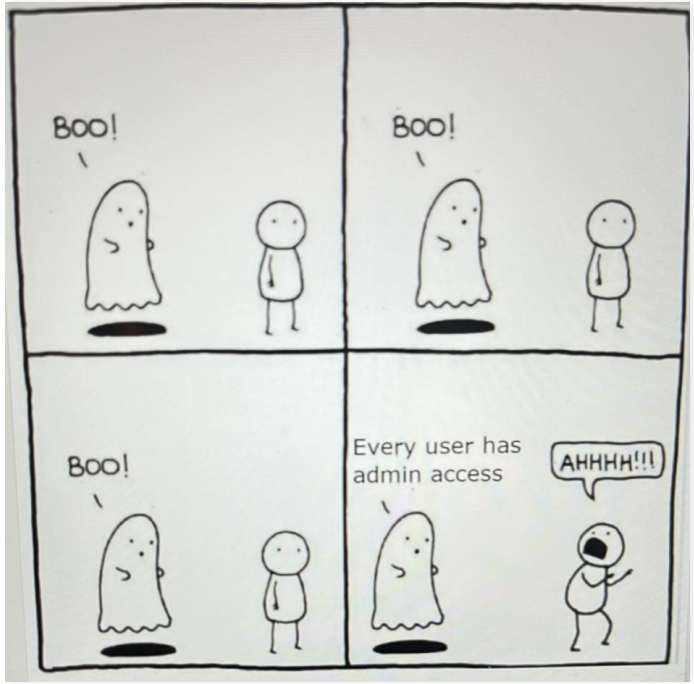


Types of Permissions

- Global permissions apply system-wide, independent of any specific resource
- Object-level permissions depend on ownership or state of a particular resource
- Role-based permissions grant access through assigned roles rather than individuals
- Action-based permissions control what operation can be performed, not just what resource is accessed

Rules of Access Control

- Permission checks must occur before side effects, not after data is touched
- **Deny-by-default** is the only sane baseline. Explicit grants beat implicit access
- Multiple permission rules compose logically and should **fail fast** on violation
- Enforcement must be centralized to avoid divergent security behavior
- Inconsistent permission evaluation is a security bug, not a code smell



Design Pitfalls

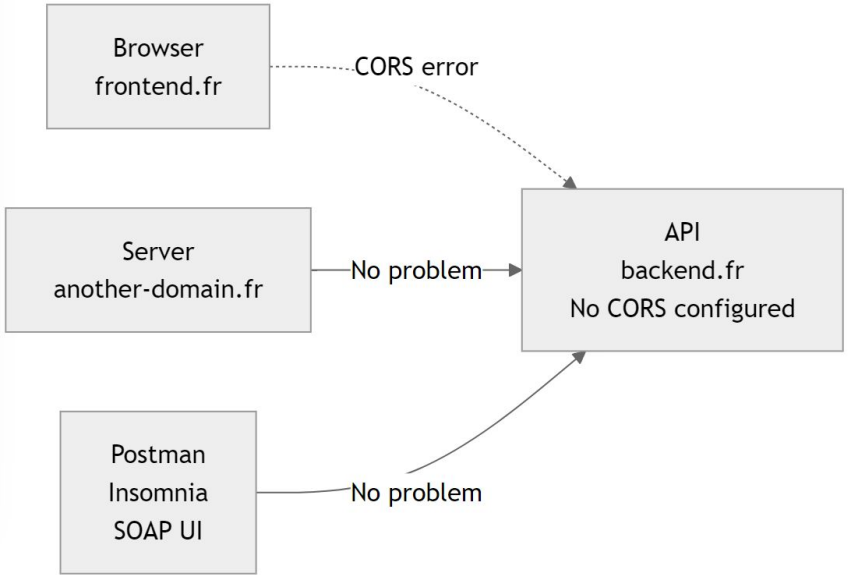
- Overloading roles until they become indistinguishable from users
- Encoding permissions in routes instead of business logic
- Mixing authentication state with authorization decisions
- Relying on client-side checks for server-side security
- Treating permission failures as edge cases instead of expected outcomes

Design Principles

- **Least privilege:** grant the minimum access required, nothing more
- **Explicitness over convention:** permissions must be readable and intentional
- **Separation of concerns:** permission logic must not leak into unrelated layers
- **Consistency** across endpoints to avoid accidental privilege escalation
- **Evolvability:** permission models must survive new features without collapse

CORS vs. Auth

- CORS: May this origin read the response?
- Authentication: Who is the requester?
- Authorization: What are they allowed to do?
- These concerns are independent and complementary
- Correct systems use all three without mixing responsibilities



CORS Flow

- Incoming request hits CORS middleware before view logic
- Middleware inspects the `Origin` header
- Response is decorated with appropriate CORS headers
- Browser decides whether to expose the response
- Django continues processing regardless of browser outcome