17 - DRF: API-Authentifizierung & Berechtigungen

Einleitung

- **Themen**: Nachdem wir eine funktionierende API erstellt haben, widmen wir uns nun ihrer Absicherung. Dieses Skript behandelt die zwei Säulen der API-Sicherheit: **Authentifizierung** ("Wer bist du?") und **Autorisierung/Berechtigungen** ("Was darfst du tun?").
- **Fokus**: Implementierung einer Token-basierten Authentifizierung, die für zustandslose (stateless) APIs der Standard ist. Anwendung von eingebauten und Erstellung von benutzerdefinierten Berechtigungsklassen, um den Zugriff auf API-Endpunkte zu steuern.
- **Lernziele**:
  - Den Unterschied zwischen Authentifizierung und Autorisierung verstehen.
  - Eine Token-Authentifizierung in DRF einrichten und anwenden.
  - Verstehen, wie ein Client (z.B. eine spätere React-App) einen Token erhält und bei Anfragen mitsendet.
  - Eingebaute DRF-Berechtigungsklassen wie `IsAuthenticated` und `IsAuthenticatedOrReadOnly` verwenden.
  - Eigene, benutzerdefinierte Berechtigungsklassen schreiben, um objektbasierte Berechtigungen zu implementieren (z.B. "Nur der Ersteller darf sein eigenes Rezept bearbeiten").
  - Berechtigungen global für das Projekt oder spezifisch pro ViewSet setzen.

1. Authentifizierung vs. Autorisierung (Die Grundlagen)

Diese beiden Begriffe werden oft verwechselt, haben aber unterschiedliche Bedeutungen:

- **Authentifizierung (Authentication)**: Der Prozess der **Identitätsprüfung**. Der Server stellt fest, *wer* der Benutzer ist. Bei APIs geschieht dies meist durch die Überprüfung eines Tokens, den der Benutzer mitsendet, nachdem er sich einmal mit Benutzername und Passwort authentifiziert hat.

- **Autorisierung (Authorization / Permissions)**: Der Prozess der **Rechteprüfung**. Nachdem die Identität des Benutzers bekannt ist, prüft der Server, ob dieser Benutzer die Berechtigung hat, die angeforderte Aktion durchzuführen (z.B. ein Rezept zu löschen).

2. Token-Authentifizierung in DRF einrichten

APIs sind zustandslos (stateless), d.h. der Server speichert keine Login-Session. Stattdessen sendet der Client bei jeder Anfrage einen "Beweis" seiner Identität mit – einen **Token**.

Der Ablauf:

1. Der Client sendet Benutzername und Passwort an einen speziellen Endpunkt (z.B. `/api/token-auth/`).
2. Der Server überprüft die Daten. Wenn sie korrekt sind, generiert er einen einzigartigen Token für diesen Benutzer und sendet ihn zurück.
3. Der Client speichert diesen Token sicher (z.B. im Local Storage des Browsers).
4. Für jede weitere Anfrage an einen geschützten Endpunkt sendet der Client den Token im `Authorization`-Header mit.
5. Der Server prüft bei jeder Anfrage die Gültigkeit des Tokens und weiß so, welcher Benutzer die Anfrage stellt.

Implementierung in DRF:

1. **`authtoken`-App hinzufügen**: Füge `'rest_framework.authtoken'` zu `INSTALLED_APPS` in `settings.py` hinzu.

```
# settings.py
INSTALLED_APPS = [
    # ...
    'rest_framework',
    'rest_framework.authtoken', # Hinzufügen
    # ...
]
```

2. **Datenbank-Tabelle erstellen**: Führe `migrate` aus, um die Tabelle zu erstellen, in der DRF die Tokens speichert.

```
python manage.py migrate
```

3. **Token-Endpunkt erstellen**: DRF bietet eine eingebaute View, um Tokens zu generieren. Binde sie in deine Haupt-`urls.py` oder eine API-spezifische `urls.py` ein.

```
# projekt/urls.py oder api/urls.py
from rest_framework.authtoken.views import obtain_auth_token

urlpatterns = [
    # ...
    path('token-auth/', obtain_auth_token, name='api_token_auth'),
]
```

An diesen Endpunkt kann ein Client nun eine `POST`-Anfrage mit `username` und `password` senden, um einen Token zu erhalten.

3. Globale Konfiguration vs. Pro-ViewSet-Konfiguration

Man kann Authentifizierungs- und Berechtigungsregeln entweder global für das gesamte Projekt oder spezifisch für einzelne ViewSets festlegen.

- **Globale Konfiguration in `settings.py`**: Dies ist nützlich, um eine Standard-Sicherheitsrichtlinie für alle API-Endpunkte festzulegen.

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated', # Standard: Kein Zugriff ohne
gültigen Token
    ]
}
```

- **Spezifische Konfiguration pro ViewSet**: Man kann die globalen Einstellungen in jedem ViewSet überschreiben.

```
# recipes/views.py
from rest_framework.permissions import AllowAny

class PublicRecipeViewSet(viewsets.ModelViewSet):
    queryset = Recipe.objects.filter(is_public=True)
    serializer_class = RecipeSerializer
    # Dieses ViewSet ignoriert die globalen Einstellungen und erlaubt jedem den Zugriff.
    permission_classes = [AllowAny]
```

4. DRF-Berechtigungsklassen (Permissions)

DRF bietet mehrere eingebaute Klassen, um gängige Berechtigungsszenarien abzudecken.

- **`AllowAny`**: Der Standard, falls nichts konfiguriert ist. Jeder (auch anonyme Benutzer) hat vollen Zugriff.
- **`IsAuthenticated`**: Nur authentifizierte Benutzer (mit gültigem Token) haben Zugriff. Anonyme Benutzer erhalten einen `401 Unauthorized` oder `403 Forbidden` Fehler.
- **`IsAdminUser`**: Nur Benutzer mit `is_staff=True` haben Zugriff.
- **`IsAuthenticatedOrReadOnly`**: Eine sehr nützliche Klasse für öffentliche APIs. Jeder (auch anonyme Benutzer) darf **lesend** zugreifen (`GET`, `HEAD`, `OPTIONS`). Schreibende Zugriffe (`POST`, `PUT`, `PATCH`, `DELETE`) sind jedoch nur für authentifizierte Benutzer erlaubt.

5. Benutzerdefinierte Berechtigungen (Custom Permissions)

Oft reichen die Standard-Berechtigungen nicht aus. Das klassische Beispiel: "Ein Benutzer darf nur seine eigenen Rezepte bearbeiten oder löschen." Dies erfordert eine objektbasierte Berechtigung.

- **Erstellen einer Custom Permission (`permissions.py`)**: Erstelle eine neue Datei `recipes/permissions.py`.

```python
# recipes/permissions.py
from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Benutzerdefinierte Berechtigung, die nur dem Besitzer eines Objekts
    erlaubt, es zu bearbeiten. Lesezugriffe sind für alle erlaubt.
    """
    def has_object_permission(self, request, view, obj):
        # Lesezugriffe (GET, HEAD, OPTIONS) sind immer erlaubt.
        if request.method in permissions.SAFE_METHODS:
            return True

        # Schreibzugriffe sind nur dem Besitzer des Objekts erlaubt.
        # Wir nehmen an, das Objekt hat ein 'author'-Attribut.
        return obj.author == request.user
```

  *`has_object_permission` wird bei Detail-Endpunkten (z.B. `/api/recipes/1/`) aufgerufen. *`SAFE_METHODS` ist eine Konstante, die `('GET', 'HEAD', 'OPTIONS')` enthält.

- **Anwendung der Custom Permission im ViewSet**:

```python
# recipes/views.py
from rest_framework.permissions import IsAuthenticatedOrReadOnly
from .permissions import IsOwnerOrReadOnly # Importieren

class RecipeViewSet(viewsets.ModelViewSet):
    queryset = Recipe.objects.all()
    serializer_class = RecipeSerializer
    # Berechtigungen werden der Reihe nach geprüft.
    permission_classes = [IsAuthenticatedOrReadOnly, IsOwnerOrReadOnly]
```

  Hier wird zuerst geprüft, ob der Benutzer für einen Schreibzugriff angemeldet ist. Wenn ja, wird als Nächstes geprüft, ob er auch der Besitzer des Objekts ist.

Fazit

- **Token-Authentifizierung**: Der Standard für zustandslose APIs. Einmal mit Login-Daten einen Token holen, dann diesen Token bei jeder Anfrage mitsenden.
- **Berechtigungsklassen**: Das mächtige Werkzeug von DRF zur Steuerung von Zugriffsrechten.
- **`IsAuthenticatedOrReadOnly`**: Eine sehr häufig verwendete Klasse für APIs, die öffentliche Lesezugriffe, aber geschützte Schreibzugriffe erfordern.
- **Custom Permissions**: Unerlässlich für objektbasierte Berechtigungen (z.B. "Benutzer darf nur eigene Daten bearbeiten"). Die Implementierung von `has_object_permission` ist hier der Schlüssel.

Projekt-Anwendung (Leitfaden-Projekt)

Die API des "Online-Umfragesystems" (`Polls`-Projekt) wird nun abgesichert.

1. **Token-Authentifizierung einrichten**: `rest_framework.authtoken` zu `INSTALLED_APPS` hinzufügen, `migrate` ausführen.
2. **Token-Endpunkt in `urls.py` hinzufügen**: Einen Pfad für `obtain_auth_token` einrichten.
3. **Globale Berechtigungen in `settings.py` setzen**:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES':
['rest_framework.authentication.TokenAuthentication'],
    'DEFAULT_PERMISSION_CLASSES': ['rest_framework.permissions.IsAuthenticatedOrReadOnly'],
}
```

4. **Testen**: Mit einem API-Tool wie Postman/Insomnia:
   - `GET`-Anfrage an `/api/polls/questions/` -> Sollte funktionieren.
   - `POST`-Anfrage an `/api/polls/questions/` ohne Authentifizierung -> Sollte fehlschlagen (401/403).
   - `POST`-Anfrage an `/api/token-auth/` mit `username` und `password` eines Testbenutzers -> Token erhalten.
   - `POST`-Anfrage an `/api/polls/questions/` mit dem Token im `Authorization`-Header (`Token <erhaltener_token>`) -> Sollte erfolgreich sein.

Cheat Sheet

DRF Auth & Permissions

- **Installation Token Auth**: `'rest_framework.authtoken'` in `INSTALLED_APPS`, dann `python manage.py migrate`.
- **Token-Endpunkt (`urls.py`)**: `path('token-auth/', obtain_auth_token)`
- **Globale Einstellungen (`settings.py`)**:

```
# REST_FRAMEWORK = {
#     'DEFAULT_AUTHENTICATION_CLASSES': [...],
#     'DEFAULT_PERMISSION_CLASSES': [...],
# }
```

- **Per-View-Einstellungen (`views.py`)**:

```
# class MyViewSet(viewsets.ModelViewSet):
#     authentication_classes = [TokenAuthentication, ...]
#     permission_classes = [IsAuthenticated, ...]
```

- **Eingebaute Berechtigungsklassen**: `AllowAny`, `IsAuthenticated`, `IsAdminUser`, `IsAuthenticatedOrReadOnly`.
- **Benutzerdefinierte Berechtigungsklasse**:

```
# from rest_framework import permissions
# class IsOwner(permissions.BasePermission):
#     def has_object_permission(self, request, view, obj):
#         return obj.author == request.user
```

Übungsaufgaben

1. **Test-API absichern**:
   - Das `Task`-API-Projekt aus der vorherigen Übung nehmen.
   - Token-Authentifizierung einrichten.
   - Das `TaskViewSet` so absichern, dass nur authentifizierte Benutzer überhaupt auf die API zugreifen können (`IsAuthenticated`).
   - Mit einem API-Tool testen, dass anonyme Anfragen fehlschlagen und Anfragen mit einem gültigen Token erfolgreich sind.
2. **Benutzerdefinierte Berechtigung schreiben**:
   - Das `Task`-Modell um einen `ForeignKey` zum `User`-Modell erweitern (`owner`).
   - Eine benutzerdefinierte Berechtigungsklasse `IsTaskOwner` schreiben, die prüft, ob `task.owner == request.user`.
   - Diese Berechtigung auf das `TaskViewSet` anwenden und testen, ob Benutzer nur ihre eigenen Aufgaben sehen und bearbeiten können.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Die API der "Community Recipe Sharing Platform" wird nun abgesichert.

Aufgabe:

1. **Token-Authentifizierung einrichten**: `rest_framework.authtoken` zu `INSTALLED_APPS` hinzufügen und `migrate` ausführen. Einen Endpunkt für den Token-Erhalt in der `api_urls.py` (oder Haupt-`urls.py`) erstellen.
2. **Standard-Berechtigungen setzen**: In `settings.py` die `DEFAULT_AUTHENTICATION_CLASSES` auf `TokenAuthentication` und die `DEFAULT_PERMISSION_CLASSES` auf `IsAuthenticatedOrReadOnly` setzen.
3. **Benutzerdefinierte Berechtigung `IsOwnerOrReadOnly` erstellen**:
   - Eine neue Datei `recipes/permissions.py` erstellen.
   - Darin die Klasse `IsOwnerOrReadOnly` wie im Beispiel oben implementieren. Sie soll prüfen, ob das Objekt ein `author`-Attribut hat, das mit `request.user` übereinstimmt.
4. **`RecipeViewSet` absichern**:
   - In `recipes/views.py` die `IsOwnerOrReadOnly`-Klasse importieren.
   - Dem `RecipeViewSet` die `permission_classes` zuweisen:

     ```
     permission_classes = [IsAuthenticatedOrReadOnly, IsOwnerOrReadOnly]
     ```

5. **Andere ViewSets absichern**:
   - Überlegen, welche Berechtigungen für `IngredientViewSet` und `StepViewSet` sinnvoll sind. `IsAuthenticatedOrReadOnly` ist hier ein guter Start (jeder darf Zutaten sehen, aber nur angemeldete Benutzer dürfen neue anlegen).
6. **Testen mit einem API-Tool (z.B. Postman)**:
   - Einen Benutzer über die Django-Admin-Oberfläche oder die Registrierungsseite anlegen.
   - **Anonym testen**: `GET`-Anfrage auf `/api/recipes/` -> Sollte funktionieren.
   - **Token holen**: `POST`-Anfrage an `/api/token-auth/` mit den Login-Daten des Benutzers. Den erhaltenen Token kopieren.
   - **Authentifiziert testen**:
     - Neue `POST`-Anfrage an `/api/recipes/` senden. Im `Authorization`-Tab "Bearer Token" oder "Token" auswählen und den Token einfügen. Der Request Body sollte das JSON für ein neues Rezept enthalten. -> Sollte funktionieren.
     - `PUT`- oder `DELETE`-Anfrage auf ein Rezept, das einem **anderen** Benutzer gehört. -> Sollte mit einem `403 Forbidden`-Fehler fehlschlagen.
     - `PUT`- oder `DELETE`-Anfrage auf ein **eigenes** Rezept. -> Sollte erfolgreich sein.