12 - Django: Benutzerauthentifizierung & Autorisierung

Einleitung

- **Themen**: Dieses Skript behandelt das eingebaute Authentifizierungssystem von Django. Es wird ein vollständiger Zyklus von Benutzer-Registrierung, Anmeldung (Login) und Abmeldung (Logout) implementiert.
- **Fokus**: Absicherung von Webanwendungen und die Verwaltung von Benutzerzugriffen.
- **Lernziele**:
  - Die Funktionsweise des Django `auth`-Systems und des `User`-Modells verstehen.
  - Eine dedizierte `accounts`-App für die Benutzerverwaltung erstellen.
  - Die eingebaute `UserCreationForm` und benutzerdefinierte Registrierungsformulare vergleichen und anwenden können.
  - Die eingebaute `AuthenticationForm` für den Login-Prozess nutzen.
  - Die zentralen Funktionen `login()`, `logout()` und `authenticate()` korrekt anwenden.
  - Den Zugriff auf Views mit dem `@login_required`-Decorator und manuellen `is_authenticated`-Prüfungen steuern.
  - Das Django Messages Framework für Benutzer-Feedback nutzen.

1. Djangos Authentifizierungssystem im Überblick

Django kommt mit einem vollständigen System für die Verwaltung von Benutzern, Gruppen und Berechtigungen. Das Herzstück ist das `User`-Modell (`django.contrib.auth.models.User`), das Felder wie `username`, `password` (sicher als Hash gespeichert), `email`, `first_name` und `last_name` enthält. Wir müssen das Rad hier nicht neu erfinden.

2. Der Workflow: Eigene `accounts`-App erstellen

Es ist eine bewährte Methode, die gesamte Logik für die Benutzerverwaltung in einer eigenen, dedizierten App zu kapseln.

1. **App erstellen**:

```
python manage.py startapp accounts
```

2. **App registrieren**: Die neue App `'accounts'` in `settings.py` zu `INSTALLED_APPS` hinzufügen.
3. **URLs einrichten**: Die URLs der `accounts`-App in der Haupt-`urls.py` des Projekts einbinden.

```
# projekt/urls.py
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('accounts.urls')),
    path('', include('recipes.urls')),
]
```

3. Registrierung (Sign-up)

Für die Registrierung gibt es zwei gängige Ansätze: Djangos eingebaute `UserCreationForm` für einen schnellen Start und ein benutzerdefiniertes Formular für mehr Flexibilität.

3.1 Methode A: Die eingebaute `UserCreationForm` (Der schnelle Weg)

Django bietet mit `django.contrib.auth.forms.UserCreationForm` ein fertiges Formular für die Benutzer-Registrierung.

- **View mit `UserCreationForm` (`accounts/views.py`)**:

```
# accounts/views.py
from django.shortcuts import render, redirect
from django.contrib.auth.forms import UserCreationForm
from django.contrib import messages
```

```
def register_simple_view(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save() # Erstellt den User und hasht das Passwort automatisch
            username = form.cleaned_data.get('username')
            messages.success(request, f'Account für {username} wurde erstellt! Bitte
anmelden.')
            return redirect('accounts:login')
    else:
        form = UserCreationForm()
    return render(request, 'accounts/register.html', {'form': form})
```

- **Vorteile**:
  - **Schnell**: Extrem wenig Code nötig, um eine funktionierende Registrierung zu implementieren.
  - **Sicher**: Die `save()`-Methode kümmert sich automatisch um das sichere Hashen des Passworts.
  - **Standardisiert**: Enthält bereits Felder für Benutzername, Passwort und Passwort-Bestätigung mit Standard-Validierungen.
- **Nachteile**:
  - **Wenig flexibel**: Es ist umständlich, Felder hinzuzufügen (z.B. E-Mail) oder das Layout und die Labels zu ändern.
  - **Keine E-Mail-Abfrage**: Standardmäßig wird nur Benutzername und Passwort abgefragt, was für viele moderne Anwendungen nicht ausreicht.

3.2 Methode B: Benutzerdefiniertes Registrierungsformular (Der flexible Weg)

Für die meisten realen Anwendungen ist ein benutzerdefiniertes Formular die bessere Wahl, da es volle Kontrolle bietet.

- **Benutzerdefiniertes Formular (`accounts/forms.py`)**:

```
# accounts/forms.py
from django import forms
from django.contrib.auth.models import User

class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput, label="Passwort")
    password2 = forms.CharField(widget=forms.PasswordInput, label="Passwort bestätigen")

    class Meta:
        model = User
        fields = ['username', 'email']

    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Passwörter stimmen nicht überein.')
        return cd['password2']
```

Die `clean_<feldname>()`-Methode wird von Django während der `is_valid()`-Prüfung automatisch aufgerufen und dient der benutzerdefinierten Validierung eines einzelnen Feldes.

- **View für benutzerdefiniertes Formular (`accounts/views.py`)**:

```
# accounts/views.py
from .forms import UserRegistrationForm
# ... (andere imports)

def register_view(request):
    if request.method == 'POST':
        form = UserRegistrationForm(request.POST)
        if form.is_valid():
```

```
            # form.save() würde den User ohne Passwort speichern. Wir müssen es hashen.
            new_user = form.save(commit=False)
            # Das Passwort sicher setzen (Hashing)
            new_user.set_password(form.cleaned_data['password'])
            new_user.save()

            # Feedback für den Benutzer geben
            messages.success(request, 'Registrierung erfolgreich! Sie können sich nun
anmelden.')

            return redirect('accounts:login') # Weiterleitung zur Login-Seite
    else:
        form = UserRegistrationForm()

    return render(request, 'accounts/register.html', {'form': form})
```

4. Anmeldung (Login)

Für den Login stellt Django ein fertiges Formular bereit: `AuthenticationForm`.

- **Login-View (`accounts/views.py`)**:

```
# accounts/views.py
from django.contrib.auth import authenticate, login
from django.contrib.auth.forms import AuthenticationForm

def login_view(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)
        if form.is_valid():
            username = form.cleaned_data.get('username')
            password = form.cleaned_data.get('password')
            user = authenticate(username=username, password=password) # Überprüft
Anmeldedaten
            if user is not None:
                login(request, user) # Erstellt die Session für den Benutzer
                return redirect('recipes:list') # Weiterleitung zur Startseite nach Login
            else:
                # Fehlermeldung, falls authenticate fehlschlägt
                messages.error(request, 'Ungültiger Benutzername oder Passwort.')
    else:
        form = AuthenticationForm()

    return render(request, 'accounts/login.html', {'form': form})
```

  - `authenticate()`: Überprüft die Anmeldedaten gegen die Datenbank. Gibt das `User`-Objekt zurück, wenn sie korrekt sind, sonst `None`.
  - `login()`: Nimmt das `request`-Objekt und das `User`-Objekt entgegen und erstellt die Benutzer-Session (setzt z.B. die Session-Cookies).

5. Abmeldung (Logout)

Die Abmeldung löscht die Session-Daten des Benutzers.

- **Logout-View (`accounts/views.py`)**:

```
# accounts/views.py
from django.contrib.auth import logout
# ...

def logout_view(request):
```

```
    logout(request)
    return redirect('recipes:list')
```

6. Views schützen (Autorisierung)

Es gibt zwei Wege, um sicherzustellen, dass nur angemeldete Benutzer eine View aufrufen können.

6.1 Der `@login_required`-Decorator

Dies ist der einfachste und gängigste Weg. Der Decorator wird vor die View-Funktion gesetzt.

```
# recipes/views.py
from django.contrib.auth.decorators import login_required

@login_required(login_url='accounts:login') # login_url ist optional, aber empfohlen
def add_recipe_view(request):
    # Logik, die nur für angemeldete Benutzer zugänglich ist
    # ...
```

Wenn ein nicht angemeldeter Benutzer diese View aufruft, wird er zur `login_url` weitergeleitet.

6.2 Manuelle Prüfung in Views mit `is_authenticated`

Manchmal soll eine View für alle zugänglich sein, aber unterschiedliche Inhalte für angemeldete und anonyme Benutzer anzeigen. Hier prüft man innerhalb der View.

```
# recipes/views.py
def recipe_list_view(request):
    # request.user ist immer verfügbar
    if request.user.is_authenticated:
        # Logik für angemeldete Benutzer
        greeting = f"Willkommen zurück, {request.user.username}!"
    else:
        # Logik für Gäste
        greeting = "Willkommen, Gast! Entdecke unsere Rezepte."

    # ... restliche Logik ...
    return render(request, 'recipes/recipe_list.html', {'greeting': greeting, ...})
```

`request.user.is_authenticated` ist eine Eigenschaft (keine Funktion), die `True` zurückgibt, wenn der Benutzer angemeldet ist, sonst `False`. Für noch detailliertere Prüfungen, z.B. ob ein Benutzer ein Mitarbeiter oder Superuser ist, siehe Abschnitt 8.

7. Das `user`-Objekt und `is_authenticated` in Templates

Dank des `auth`-Kontextprozessors ist das `user`-Objekt in jedem Template verfügbar, das mit `render()` gerendert wird.

- **Bedingte Inhalte im Template anzeigen**: Der häufigste Anwendungsfall ist die Navigation, aber es kann für jeden Inhalt verwendet werden, der je nach Anmeldestatus variieren soll.

```
{# base.html #}
<nav>
    <a href="{% url 'recipes:list' %}">Alle Rezepte</a>

    {% if user.is_authenticated %}
        <a href="{% url 'recipes:add_recipe' %}">Rezept hinzufügen</a>
        <span>Hallo, {{ user.username }}!</span>
        <a href="{% url 'accounts:logout' %}">Abmelden</a>
    {% else %}
        <a href="{% url 'accounts:login' %}">Anmelden</a>
        <a href="{% url 'accounts:register' %}">Registrieren</a>
```

```
        {% endif %}
    </nav>
```

8. Detaillierte Benutzerstatus-Prüfungen

Neben dem reinen Anmeldestatus bietet Djangos `User`-Modell weitere nützliche Boolean-Eigenschaften, um den Status und die Berechtigungen eines Benutzers zu prüfen.

8.1 Anonymer Benutzer: `is_anonymous` vs. `not is_authenticated`

Wenn ein Benutzer nicht angemeldet ist, repräsentiert Django ihn durch ein spezielles `AnonymousUser`-Objekt.

- **`user.is_anonymous`**: Diese Eigenschaft ist `True` für Gäste und `False` für alle angemeldeten Benutzer. Sie ist das direkte Gegenstück zu `is_authenticated`.
- **`not user.is_authenticated`**: Dies ist die logische Verneinung und führt zum exakt gleichen Ergebnis wie `user.is_anonymous`. Die Wahl ist oft eine Frage der Lesbarkeit im Code.

Beispiel im Template:

```
{% if user.is_anonymous %}
    <p>Als Gast können Sie unsere öffentlichen Rezepte ansehen. <a href="{% url
'accounts:register' %}">Registrieren</a> Sie sich, um eigene Rezepte zu erstellen.</p>
{% endif %}
```

8.2 Berechtigungs-Flags: `is_staff` und `is_superuser`

Diese Flags sind entscheidend für die Zugriffskontrolle auf administrative Funktionen.

- **`user.is_staff`**:
    - **Zweck**: Bestimmt, ob ein Benutzer auf die Django-Admin-Oberfläche (`/admin/`) zugreifen darf.
    - **Verwendung**: Nützlich für Teammitglieder oder Redakteure, die Inhalte verwalten sollen, aber keine vollen Systemrechte benötigen. Ein "Staff"-Benutzer hat nur die Berechtigungen, die ihm explizit zugewiesen wurden.
- **`user.is_superuser`**:
    - **Zweck**: Gewährt einem Benutzer alle Berechtigungen im System, ohne dass sie einzeln zugewiesen werden müssen. Ein Superuser hat vollen Zugriff auf den Admin-Bereich und ignoriert alle Berechtigungsprüfungen.
    - **Verwendung**: Für den Hauptadministrator der Webseite. Der mit `python manage.py createsuperuser` erstellte Benutzer ist immer ein Superuser.

Hierarchie: Jeder Superuser ist automatisch auch ein "Staff"-Mitglied (`is_staff` ist `True`), aber nicht umgekehrt.

Beispiel für Berechtigungen im Template:

```
{% if user.is_authenticated %}
    <div class="admin-links">
        {% if user.is_superuser %}
            <a href="/admin/">Voller Admin-Zugriff</a>
        {% elif user.is_staff %}
            <a href="/admin/">Zum Admin-Bereich</a>
        {% endif %}
    </div>
{% endif %}
```

8.3 Aktivitäts-Flag: `is_active`

- **Zweck**: Dies ist ein entscheidender Schalter. Nur Benutzer mit `is_active=True` können sich erfolgreich authentifizieren und anmelden.

- **Verwendung**: Um einen Benutzer zu sperren oder zu deaktivieren, ohne sein Konto zu löschen. Setzt man dieses Flag im Admin-Bereich auf `False`, kann sich der Benutzer nicht mehr anmelden. Standardmäßig ist jeder neue Benutzer aktiv.

In einer View muss man dies selten manuell prüfen, da Djangos `authenticate()`-Funktion dies bereits im Hintergrund berücksichtigt.

Fazit

- **`UserCreationForm` vs. Custom Form**: `UserCreationForm` ist schnell für Prototypen; ein eigenes `ModelForm` ist flexibler für reale Anwendungen.
- **Auth-Funktionen**: `authenticate()` (prüft), `login()` (startet Session), `logout()` (beendet Session) sind die zentralen Bausteine.
- **Zugriffskontrolle**: Der `@login_required`-Decorator ist der Standard zum Schutz ganzer Views. Die `user.is_authenticated`-Eigenschaft ermöglicht feinere Logik in Views und Templates.
- **User-Objekt**: Ist über `request.user` in Views und `user` in Templates universell verfügbar.

Projekt-Anwendung (Leitfaden-Projekt)

Für das "Online-Umfragesystem" (`Polls`-Projekt) wird ein vollständiger Authentifizierungs-Workflow implementiert.

1. **`accounts`-App erstellen**, registrieren und URLs einbinden.
2. **`UserCreationForm`** für eine schnelle Registrierung in einer `register_simple_view` implementieren.
3. **Views und Templates** für Login und Logout erstellen.
4. **Basis-Template `base_polls.html` anpassen**, um die Navigation basierend auf `user.is_authenticated` anzuzeigen und einen Bereich für Django Messages hinzuzufügen.
5. **View `add_question_modelform`** mit `@login_required` schützen.

Cheat Sheet

Wichtige Auth-Funktionen (`django.contrib.auth`)

- `authenticate(request, username='...', password='...')`: Überprüft Anmeldedaten.
- `login(request, user)`: Startet die Session.
- `logout(request)`: Beendet die Session.

Formulare

- `UserCreationForm`: Built-in Formular für schnelle Registrierung.
- `AuthenticationForm`: Built-in Formular für den Login.

Zugriffskontrolle & User-Objekt

- `@login_required(login_url='...')`: Decorator zum Schutz von Views.
- `request.user`: Das `User`-Objekt in Views.
- `user.is_authenticated`: Boolean-Eigenschaft in Views (`request.user.is_authenticated`) und Templates (`user.is_authenticated`).
- `LOGIN_URL = 'url_name'` (in `settings.py`).

Messages Framework (`django.contrib`)

- `messages.success(request, 'Nachricht')`, `messages.error(...)` etc.

Übungsaufgaben

1. **Registrierungsformular anpassen**:
   - Das benutzerdefinierte `UserRegistrationForm` so erweitern, dass auch `first_name` und `last_name` abgefragt werden.
2. **Geschützte Profilseite erstellen**:
   - Eine neue View `user_profile_view` erstellen, die `user.email`, `user.first_name` und `user.last_name` des angemeldeten Benutzers anzeigt.

- Diese View mit `@login_required` schützen.
- Eine URL dafür einrichten und in die Navigation des Basis-Templates einbauen (nur sichtbar, wenn `user.is_authenticated`).

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Die "Community Recipe Sharing Platform" erhält nun eine vollständige Benutzerverwaltung.

Aufgabe:

1. **`accounts`-App erstellen**: Eine neue App `accounts` im Projekt erstellen, registrieren und in die Haupt-`urls.py` einbinden.
2. **Authentifizierungs-Workflow implementieren**:
   - In der `accounts`-App das benutzerdefinierte `UserRegistrationForm`, die Views (`register_view`, `login_view`, `logout_view`) und Templates (`register.html`, `login.html`) erstellen.
3. **Basis-Template `base_recipes.html` anpassen**:
   - Eine Navigation hinzufügen, die basierend auf `{% if user.is_authenticated %}` unterschiedliche Links anzeigt (Login/Registrieren vs. Logout/Rezept hinzufügen/Profil).
   - Den Code-Block zur Anzeige von Django Messages einfügen.
4. **Views schützen**:
   - Die Views zum Hinzufügen (`add_recipe_view`) und Bearbeiten (`edit_recipe_view`) von Rezepten mit dem `@login_required`-Decorator schützen.
5. **Autor zuweisen**:
   - In der `add_recipe_view` sicherstellen, dass die `form.save(commit=False)`-Logik den angemeldeten Benutzer als Autor des Rezepts setzt: `recipe.author = request.user`.
6. **Zugriff in `edit_recipe_view` beschränken**:
   - Innerhalb der `edit_recipe_view` (nach dem Holen der `recipe_instance`) eine `if`-Abfrage einbauen, die prüft, ob `recipe_instance.author == request.user`. Wenn nicht, soll der Zugriff verweigert werden.
7. **Testen**: Den kompletten Zyklus testen: Registrieren, Anmelden, Rezept erstellen, Rezept bearbeiten, Abmelden. Versuchen, als nicht angemeldeter oder falscher Benutzer auf geschützte Seiten zuzugreifen.