Authentication System

> Django Authentication System
>
> Every Django application comes with a complete and flexible authentication system.
>
> ```
> INSTALLED_APPS = [
>         'django.contrib.auth',
> ]
> ```
>
> Using this system we can
>
> - Identify Users
> - Login
> - Logout
> - Change their password
> - ...
>
> ---
>
> ## Models
>
> Some of the models in this app
>
> - User
> - Group
> - Permission
> - And their combinations
>
> And for each of these models we have persistent tables
>
> ---
>
> ## User Model
>
> - id

- password - *encrypted*
- Last Login
- is superuser
- username
- first name
- last name
- email
- is staff - *admin area* (Can be controlled)
- is active
- date joined
  We can easily add columns to this table (Will be done...)

Authentication Middleware

We also have a `MIDDLEWARE` for authentication

```
MIDDLEWARE = [

'django.contrib.auth.middleware.AuthenticationMiddleware'
]
```

This middleware reads user information from the request and sets the `user` attribute on the request object.

Every request object at runtime is going to have an attribute called `request.user` this is either set to:

- An instance of the `AnonymousUser` class
- An Actual User object
  This is the job of this middleware

Customizing the User Model

Sometimes we need to store additional data for a user. Here we have two options

- Using **Inheritance** to extend the User Model
- Creating a Profile Model and add a one-to-one link to the User Model - **Composition**

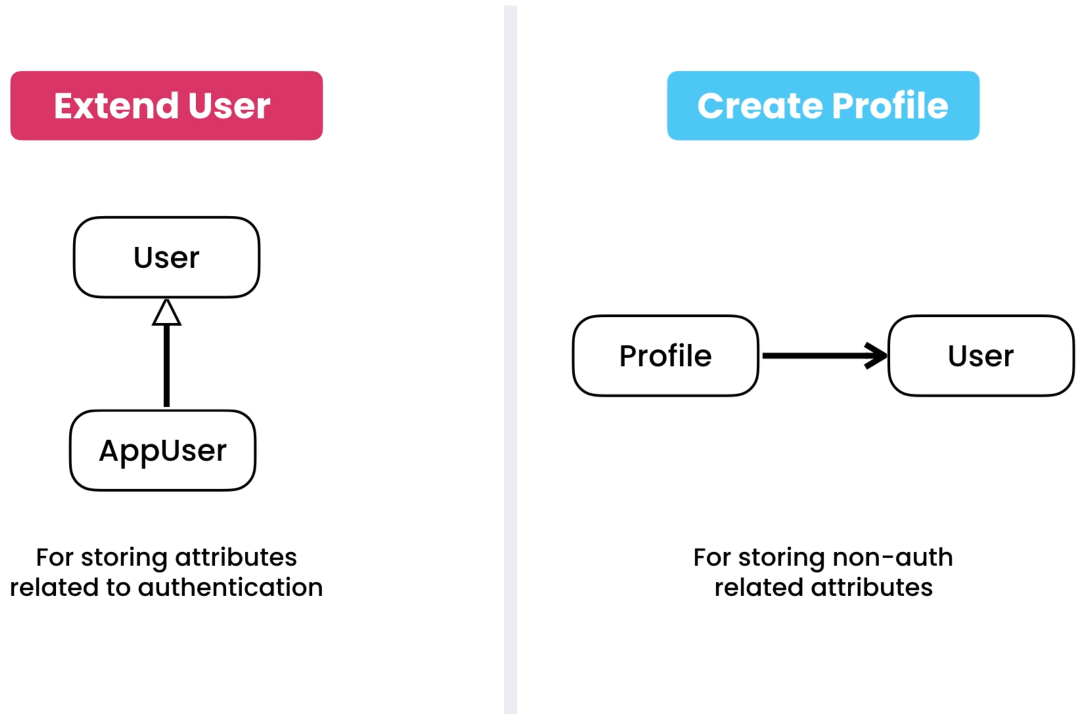
Database Terms

First approach results in the extending of the `auth_user` table. meaning we add columns to this table.

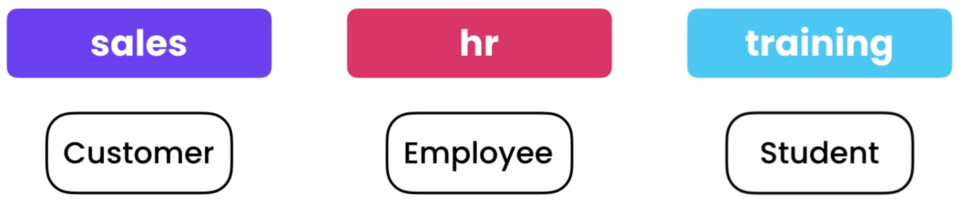
With the second approach, the `auth_user` is untouched but we are going to create a new table and include all the additional fields in that table. Then we're going to have a one-to-one relationship between these tables.

So practically speaking, the `User` table should only contain fields that are used for authentication and the first approach is only ideal if we want to add columns related to authentication. But anything extra that is not related to authentication should be done using the **Profile** approach:

- Birthdate
- Addresses
- etc.


Extend User
User
AppUser
For storing attributes related to authentication
Create Profile
Profile
User
For storing non-auth related attributes

with the second approach, we allow each app to have a different concept of a users **Profile**.



Limitations

Using the first approach (Extending the `User` model) cannot be done in the middle of a project. It can be done but it is tricky. So most of the times we use the second approach to define different schemas for different apps.

Extending the User Model

In the `auth_user` table we have a unique constraint on the username column. But let's say we want to allow our users to login using their email as well. Currently the email field does not have a unique constraint. This is a change that involves authentication so it calls for Extending the User Model.

- Create a new user model that derives from `AbstractUser` model defined in `auth` app
- Set `AUTH_USER_MODEL` in the `settings` module to your custom user model - **From this point onward you never reference the user model directly.**
- Modify the `AdminUser` model to match your schema using `fieldsets` and other properties

Create a New User Model

- Should not be created in the `store` app because

`store_custom` is a good candidate because it's very specific to this project and it's where we combine features from different apps. A better name for this app would be `core`.

Renaming steps

- Rename the directory using refactoring
- Go to the apps module in the `core` folder
- Refactor the class name from `StoreCustomConfig` to `CoreConfig`
- Change the name attribute to `core`
- Change app name in `settings.py`

User Model Creation

go in the models module

```python
from django.db import models
from django.contrib.auth.models import AbstractUser


class User(AbstractUser):
    email = models.EmailField(unique=True)
```

Change the default authentication model

go to `settings.py`

```python
AUTH_USER_MODEL = 'core.User'
```

Issues

1. `LikedItem` references the old user model

```python
from django.conf import settings


class LikedItem(models.Model):
    user = models.ForeignKey(
            settings.AUTH_USER_MODEL, # No Coupling
            on_delete=models.CASCADE
        )
    ...
```

2. `Dependency on app with no migrations` - Solution: Recreate the database
    1. `makemigrations`
    2. `migrate` -> Migration `admin.0001_initial` is applied before its dependency `core.0001_initial` on database

> This is exactly why we cannot easily extend the user model in the middle of a project.

BEST PRACTICE

Always create a custom user model at the beginning of your project. Even if you don't want to change anything.

```python
class User(AbstractUser):
        pass # No changes
```

Modifying the `UserAdmin`

Registering the new Model

If you login the admin panel, you can see that after changing the `AUTH_USER_MODEL` attribute, we can no longer see the `user` below the auth app. So we need to **Register the new User Model in the admin panel**.

in `core/admin.py`

```python
from django.contrib.auth.admin import UserAdmin as
BaseUserAdmin

@admin.register(User)
class UserAdmin(BaseUserAdmin):
        pass
```

Modifying the `fieldset`

Currently the form for adding a user does not quite match our database schema:

- the Email field is not shown. meaning it sends blank (null) to the database and because we added a unique constraint, this is going to be an issue
- I want to add the first and last name to the user creation form. even though they are not required in order to create a user.

> These changes can be done using `add_fieldset` in the new `UserAdmin` class

```python
@admin.register(User)
class UserAdmin(BaseUserAdmin):
    add_fieldsets = (
        (
            None,
            {
                "classes": ("wide",),
                "fields": (
                    "username",
                    "password1",
                    "password2",
                    "email",
                    "first_name",
                    "last_name"
                ),
            },
        ),
    )
```

Tip

Copy the `add_fieldset` from the Base Class and change it to your liking.

Creating User Profiles

To define a User Profile we first create a profile model like `Customer` and in the profile model we add a `one-to-one` relationship with the `AUTH_USER_MODEL`

```
class Customer(models.Model):
    MEMBERSHIP_BRONZE = 'B'
    MEMBERSHIP_SILVER = 'S'
    MEMBERSHIP_GOLD = 'G'

    MEMBERSHIP_CHOICES = [
        (MEMBERSHIP_BRONZE, 'Bronze'),
        (MEMBERSHIP_SILVER, 'Silver'),
        (MEMBERSHIP_GOLD, 'Gold'),
    ]
    phone  = models.CharField(max_length=255)
    birth_date = models.DateField(null=True, blank=True)
    membership = models.CharField(
        max_length=1, choices=MEMBERSHIP_CHOICES,
default=MEMBERSHIP_BRONZE)
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE)

    @admin.display(ordering='user__first_name')
    def first_name(self):
        return self.user.first_name

    @admin.display(ordering='user__last_name')
    def last_name(self):
        return self.user.last_name

    def __str__(self):
        return f'{self.user.first_name}
{self.user.last_name}'

    class Meta:
        ordering = ['user__first_name', 'user__last_name']
```

- Remove Redundant fields in this case: first, last name and email
- Tune it more for admin panel functionality

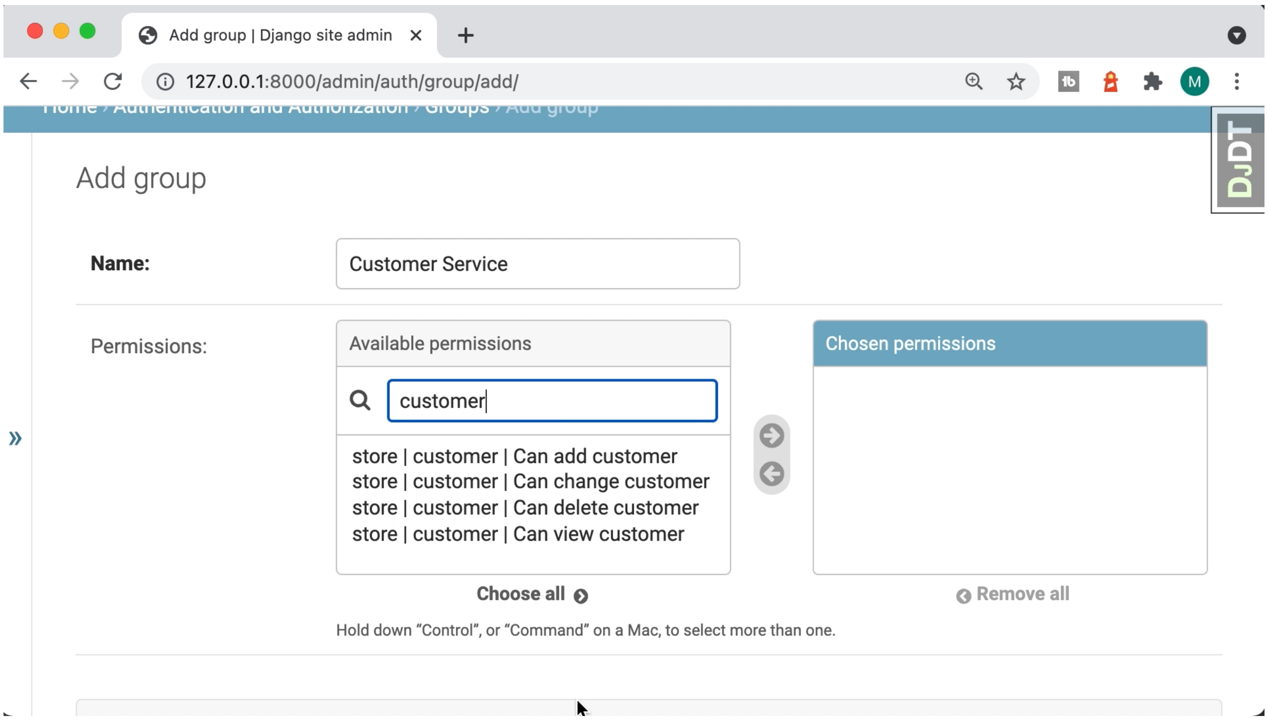
```
@admin.register(models.Customer)
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name',
'membership', 'orders']
    list_editable = ['membership']
    list_per_page = 10
    list_select_related = ['user']
    autocomplete_fields = ['user']
    ordering = ['user__first_name', 'user__last_name']
    search_fields = ['user__first_name__istartswith',
'user__last_name__istartswith']

    @admin.display(ordering='orders_count')
    def orders(self, customer):
        url = (
                reverse('admin:store_order_changelist') +
'?' +
                urlencode({'customer__id':
str(customer.id)})
        )
        return format_html('<a href="{}">{} Orders</a>',
url, customer.orders_count)

    def get_queryset(self, request):
        return super().get_queryset(request).annotate(
            orders_count=Count('order')
        )
```

Groups and permissions

Group is a collection of permissions

- instead of assigning users to multiple permissions we define a group with all its permissions and add the user to that group(s)

Each time we create a new model, Django will automatically create some permissions like this


Add group | Django site admin
127.0.0.1:8000/admin/auth/group/add/
DJDT

Add group

Name: Customer Service

Permissions:

Available permissions

customer

store | customer | Can add customer
store | customer | Can change customer
store | customer | Can delete customer
store | customer | Can view customer

Chosen permissions

Choose all

Remove all

Hold down “Control”, or “Command” on a Mac, to select more than one.

These permissions use the `auth_permission` table.

After creating a new group we can assign a user to it. We can also assign the user some explicit permission outside the group.

Creating Custom Permissions

Sometimes we have operations that are not quite about creating, updating, deleting or viewing data. for example think of canceling an order. It's a special kind of update. We just want to change its status to canceled. And let's say we want to give the ability to cancel but not delete or update an order completely to someone. This is where we create custom permissions.

in the models module

> ```
> class Order(models.Model):
>         class Meta:
>     permissions = [
>             # Code Name (UNIQUE) , Description
>         ('cancel_order', 'can cancel order')
>     ]
> ```
>
> then we make and run migrations
> The actual functionality comes later ...