Custom User Model Django

what is custom user model in django

In Django, the custom user model refers to a user model that you define yourself, rather than using Django's default `User` model. By default, Django provides a `User` model with fields like `username`, `email`, `password`, etc. However, in many cases, you may need to extend or modify this default behavior to better fit the needs of your application.

Why Use a Custom User Model?

- **Custom Fields:** You may want to add extra fields to your user model, such as a `date_of_birth`, `profile_picture`, or `bio`.
- **Change Field Types:** For example, using an email address as the unique identifier instead of a username.
- **Different Authentication Methods:** You might want to use a different authentication method (like phone numbers) instead of the default username and password.
- **Better Control:** It provides more flexibility and control over user-related data and behavior.

How to Create a Custom User Model

1. **Create a Custom User Model:**
   - You can create a custom user model by subclassing `AbstractUser` or `AbstractBaseUser` from `django.contrib.auth.models`.
   - `AbstractUser` includes all the fields and methods of the default user model, and you can add or modify fields.
   - `AbstractBaseUser` provides only the core authentication fields and methods, which gives you more control but requires more work.

   Example with `AbstractUser`:

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

   class CustomUser(AbstractUser):
       date_of_birth = models.DateField(null=True, blank=True)
   ```

   Example with `AbstractBaseUser`:

   ```python
   from django.contrib.auth.models import AbstractBaseUser, BaseUserManager
   from django.db import models
   ```

```
    class CustomUserManager(BaseUserManager):
        def create_user(self, email, password=None, **extra_fields):
            if not email:
                raise ValueError("The Email field must be set")
            email = self.normalize_email(email)
            user = self.model(email=email, **extra_fields)
            user.set_password(password)
            user.save(using=self._db)
            return user

        def create_superuser(self, email, password=None, **extra_fields):
            extra_fields.setdefault('is_staff', True)
            extra_fields.setdefault('is_superuser', True)

            return self.create_user(email, password, **extra_fields)

    class CustomUser(AbstractBaseUser):
        email = models.EmailField(unique=True)
        first_name = models.CharField(max_length=30, blank=True)
        last_name = models.CharField(max_length=30, blank=True)
        is_active = models.BooleanField(default=True)
        is_staff = models.BooleanField(default=False)

        objects = CustomUserManager()

        USERNAME_FIELD = 'email'
        REQUIRED_FIELDS = ['first_name', 'last_name']
```

2. **Update Settings:**
   - After creating the custom user model, you need to update the `AUTH_USER_MODEL` setting in your Django project's `settings.py` file.

```
python

  AUTH_USER_MODEL = 'yourapp.CustomUser'
```

3. **Migrate the Database:**
   - If you are starting a new project, you can just run the migration commands.
   - If you are modifying an existing project, you may need to take extra steps to migrate the data from the old user model to the new one.

```
bash

  python manage.py makemigrations
  python manage.py migrate
```

Best Practices

- **Create the custom user model at the start of your project** if you anticipate needing custom fields or functionality. It's much easier to create the model initially than to change it later.
- **Always use `get_user_model()`** when referring to the user model in your code. This ensures that you are using the correct user model, whether it's the default or a custom one.

```
python

  from django.contrib.auth import get_user_model
```

```
User = get_user_model()
```

- **Avoid relying on the default `User` model after you've defined a custom one** to avoid potential conflicts.

Conclusion

Creating a custom user model in Django gives you the flexibility to tailor user authentication and profiles to meet your application's needs. While it requires a bit more setup, it can save you from headaches down the road if you need to add or change user-related fields and functionality.

differnce bw abstractuser and abstractbaseuser ?

`AbstractUser` and `AbstractBaseUser` are both abstract base classes provided by Django to help you create a custom user model, but they serve different purposes and offer different levels of flexibility. Here's a breakdown of their differences:

1. AbstractUser

- **Purpose:** `AbstractUser` is a full-featured, built-in user model that includes all the fields and methods of Django's default `User` model. It is designed to be a convenient starting point if you only need to add a few custom fields or make minor modifications.
- **Fields:** Inherits the following fields from Django's default `User` model:
  - `username`
  - `email`
  - `first_name`
  - `last_name`
  - `password`
  - `groups`
  - `user_permissions`
  - `is_staff`
  - `is_active`
  - `is_superuser`
  - `last_login`
  - `date_joined`
- **Usage:** Use `AbstractUser` when you want to extend the default user model by adding additional fields or slightly modifying existing behavior, while keeping most of Django's default authentication and user management features.
- **Example:**

```
python

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

```
class CustomUser(AbstractUser):
    date_of_birth = models.DateField(null=True, blank=True)
```

- **Customizations:** You can add new fields or override methods, but you retain most of the default behavior.

2. AbstractBaseUser

- **Purpose:** `AbstractBaseUser` is a more bare-bones class that provides only the core authentication functionality (e.g., password handling). It does not include fields like `username`, `email`, `first_name`, or `last_name`. This class is designed for situations where you want full control over the user model and need to define your own fields and behavior.
- **Fields:** Only includes:
  - `password`
  - `last_login` (optional, can be omitted if you don't need to track it)
- **Usage:** Use `AbstractBaseUser` when you need to completely customize the user model, including which fields are used to identify users (e.g., using email or phone number instead of a username) and how authentication is handled. This option requires you to write more code, but it provides maximum flexibility.
- **Example:**

```python
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager
from django.db import models

class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError("The Email field must be set")
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)

        return self.create_user(email, password, **extra_fields)

class CustomUser(AbstractBaseUser):
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=30, blank=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = CustomUserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name']
```

- **Customizations:** You must define your own fields (like `email`, `username`, etc.), user manager class, and authentication methods. You have to specify which field should be used for authentication (`USERNAME_FIELD`).

Key Differences

- **Level of Control:**
  - `AbstractUser`: Higher-level, includes many built-in fields and methods, less control needed.
  - `AbstractBaseUser`: Lower-level, only core authentication, more control needed.
- **Fields Included:**
  - `AbstractUser`: Includes common fields like `username`, `email`, `first_name`, etc.
  - `AbstractBaseUser`: Only includes `password` and optionally `last_login`.
- **Ease of Use:**
  - `AbstractUser`: Easier to use if you want to extend the default user model with minimal changes.
  - `AbstractBaseUser`: Requires more setup but provides maximum customization.

In summary, if you need to make minor adjustments to the default user model, go with `AbstractUser`. If you need to create a custom user model from scratch, use `AbstractBaseUser`.

also show the visual representation of abstractuser and abstractbaseuser

The image has expired. Refresh ChatGPT page and retry saving to PDF.

Here is the visual representation comparing Django's `AbstractUser` and `AbstractBaseUser` models. The diagram illustrates the different fields associated with each and how `AbstractUser` is built on top of `AbstractBaseUser`.

i mean give the example with code and show the field of visualizable which can i see

To visualize the differences between `AbstractUser` and `AbstractBaseUser` with code examples, I'll provide code snippets for each model and explain how their fields would appear visually.

1. Using `AbstractUser`:

When using `AbstractUser`, you inherit all the fields and methods from Django's default user model. Here's an example of how you might define a custom user model using `AbstractUser`:

Code Example:

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

class CustomUser(AbstractUser):
    date_of_birth = models.DateField(null=True, blank=True)
    profile_picture = models.ImageField(upload_to='profile_pics/', null=True, blank=True)
```

Fields Included:

- `username`
- `email`
- `first_name`
- `last_name`
- `password`
- `groups`
- `user_permissions`
- `is_staff`
- `is_active`
- `is_superuser`
- `last_login`
- `date_joined`
- **Custom Fields:**
  - `date_of_birth`
  - `profile_picture`

Visual Representation:

```vbnet
CustomUser (inherits from AbstractUser)
├── username
├── email
├── first_name
├── last_name
├── password
├── groups
├── user_permissions
├── is_staff
├── is_active
├── is_superuser
├── last_login
├── date_joined
```

```
    ├── date_of_birth (custom field)
    └── profile_picture (custom field)
```

2. Using `AbstractBaseUser`:

When using `AbstractBaseUser`, you must define the fields and behavior yourself. This gives you more flexibility but requires more work.

Code Example:

python

```python
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager
from django.db import models

class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError("The Email field must be set")
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        return self.create_user(email, password, **extra_fields)

class CustomUser(AbstractBaseUser):
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=30, blank=True)
    date_of_birth = models.DateField(null=True, blank=True)
    profile_picture = models.ImageField(upload_to='profile_pics/', null=True, blank=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = CustomUserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name']
```

Fields Included:

- `email`
- `first_name`
- `last_name`
- `password` (inherited from `AbstractBaseUser`)
- `last_login` (inherited from `AbstractBaseUser`)
- `date_of_birth`
- `profile_picture`
- `is_active`
- `is_staff`

Visual Representation:

```vbnet
CustomUser (inherits from AbstractBaseUser)
├── email
├── first_name
├── last_name
├── password (inherited from AbstractBaseUser)
├── last_login (inherited from AbstractBaseUser)
├── date_of_birth
├── profile_picture
├── is_active
└── is_staff
```

Summary of Differences:

- **AbstractUser**: Includes a full set of fields (`username`, `email`, `first_name`, etc.) with additional custom fields on top.
- **AbstractBaseUser**: Only includes core fields (`password`, `last_login`), requiring you to define everything else, providing more flexibility but more setup.

In the visual representation, `AbstractUser` will show more fields by default, while `AbstractBaseUser` will have fewer fields, with the potential to fully customize which fields are included.

is it neccessary class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError("The Email field must be set")
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        return self.create_user(email, password, **extra_fields)

Yes, defining a custom user manager like `CustomUserManager` is necessary when you use `AbstractBaseUser`. Here's why:

Why You Need a Custom User Manager

- **Custom User Creation:** The default user manager (`UserManager`) is designed for Django's built-in `User` model, which relies on specific fields like `username`. When you define a custom user model using `AbstractBaseUser`, you likely change the fields used for user identification (e.g.,

using `email` instead of `username`). Therefore, you need a custom user manager to handle the creation and management of users according to your custom fields.
- **Password Handling:** The `create_user` and `create_superuser` methods in the user manager ensure that passwords are properly hashed before being saved to the database.
- **Superuser Creation:** The `create_superuser` method is crucial for creating superuser accounts with elevated permissions. Without this method, you won't be able to create superusers through Django's management commands (`createsuperuser`).

Key Responsibilities of `CustomUserManager`:

1. **Handling Required Fields:**
   - The manager checks if essential fields (like `email`) are provided when creating a user.
2. **Normalizing Data:**
   - The `normalize_email` method ensures consistency in email storage.
3. **Setting Passwords:**
   - The manager handles password hashing via the `set_password` method.
4. **Defining Superuser Attributes:**
   - The `create_superuser` method sets necessary attributes like `is_staff` and `is_superuser` to `True`.

What Happens Without a Custom User Manager?

If you don't define a custom user manager when using `AbstractBaseUser`:

- **User Creation Will Fail:** Django won't know how to create or manage your custom user instances because the built-in manager is not designed to work with your custom model.
- **Superuser Creation Will Fail:** The `createsuperuser` command relies on the `create_superuser` method, so without it, you won't be able to create superusers.

Example of Custom User Manager in Context:

Here's how the custom user manager fits into the custom user model:

```python
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager
from django.db import models

class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError("The Email field must be set")
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        return self.create_user(email, password, **extra_fields)

class CustomUser(AbstractBaseUser):
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=30, blank=True)
```

```
    date_of_birth = models.DateField(null=True, blank=True)
    profile_picture = models.ImageField(upload_to='profile_pics/', null=True, blank=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = CustomUserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name']
```

In summary, the custom user manager is necessary when using `**AbstractBaseUser**` because it ensures that your custom user model can be properly created, managed, and authenticated within Django's framework.