**Game Title: PokéTrade Quest**

**Overview:**

A web-based platform where players can trade Gen 1 Pokémon, aiming to collect all 151 Pokémon and level them up to 100. Players start with 10 free Pokémon and can gain experience and rewards through trading.

**Features:**

**1. User Account System:**

* Players create accounts to track their Pokémon collection and levels.
* Profiles show collected Pokémon, levels, and trading history.

**2. Starting Pokémon:**

* Each new player receives a randomized set of 10 Pokémon from the Gen 1 roster.

**3. Trading System:**

* **Trade Offers:** Players can list Pokémon for trade, set desired Pokémon in exchange, and negotiate with others.
* **Market Trends:** A visual display of which Pokémon are in demand, influencing trading strategies.
* **Trading Limits:** Players can only trade a certain number of Pokémon per day to encourage thoughtful trades.

**4. Experience System:**

* Players earn experience points (XP) based on:
  + **Buying Pokémon:** 5 XP for each Pokémon bought.
  + **Selling Pokémon:** 10 XP for each Pokémon sold.
* Additional XP for trading rare Pokémon or completing collections.

**5. Leveling Up Pokémon:**

* Players can level up their Pokémon through battles (simulated or against AI) or by trading.
* A simple battle system allows players to use their Pokémon against NPC trainers to gain experience.

**6. Collection Goals:**

* Track progress toward collecting all 151 Pokémon.
* Special badges or rewards for completing certain milestones (e.g., first to complete the Pokédex).

**7. Events and Challenges:**

* Weekly events where certain Pokémon are featured, offering bonus XP or exclusive Pokémon.
* Seasonal challenges that encourage trading and battling to earn unique rewards.

**8. Community Features:**

* Forums or chat rooms for players to discuss trades, strategies, and share tips.
* A leaderboard showing top traders and collectors.

**Monetization (Optional):**

* **Premium Memberships:** Offers perks like more trading slots, exclusive Pokémon, or bonus XP.
* **In-game Currency:** Players can purchase cosmetic items or special Pokémon with real money.

**Design Elements:**

* **User-Friendly Interface:** Clean layout with easy navigation for trading, profiles, and events.
* **Pokémon Artwork:** Use official or fan art for aesthetic appeal.

**Additional Considerations:**

* Ensure there’s a fair trading system to prevent exploits (e.g., trading "fake" Pokémon).
* Include tutorials for new players to understand trading and leveling mechanics.

This concept could create an engaging community of Pokémon fans while incorporating trading mechanics that encourage interaction and strategy. What do you think?

Implementing a database for your Pokémon trading website using Django is a great choice due to its powerful ORM and built-in admin interface. Here’s how you can structure your database and implement it effectively.

**Database Design**

1. **Models (Tables)**: Here are the key models you’ll need to represent the core features of your game.

User Model

from django.contrib.auth.models import User

from django.db import models

class Profile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

experience = models.IntegerField(default=0)

total\_trades = models.IntegerField(default=0)

Pokémon Model

class Pokemon(models.Model):

name = models.CharField(max\_length=50)

level = models.IntegerField(default=1)

owner = models.ForeignKey(Profile, related\_name='pokemons', on\_delete=models.CASCADE)

is\_tradeable = models.BooleanField(default=True)

# Add other attributes as needed (e.g., type, stats)

Trade Model

class Trade(models.Model):

seller = models.ForeignKey(Profile, related\_name='sales', on\_delete=models.CASCADE)

buyer = models.ForeignKey(Profile, related\_name='purchases', on\_delete=models.CASCADE, null=True, blank=True)

pokemon = models.ForeignKey(Pokemon, on\_delete=models.CASCADE)

trade\_date = models.DateTimeField(auto\_now\_add=True)

is\_completed = models.BooleanField(default=False)

Market Model (for tracking trades)

class Market(models.Model):

pokemon = models.ForeignKey(Pokemon, on\_delete=models.CASCADE)

price = models.DecimalField(max\_digits=10, decimal\_places=2)

listed\_date = models.DateTimeField(auto\_now\_add=True)

listed\_by = models.ForeignKey(Profile, on\_delete=models.CASCADE)

**Relationships**:

* **Profile** to **User**: One-to-One relationship for user accounts.
* **Profile** to **Pokemon**: One-to-Many relationship; each profile can own multiple Pokémon.
* **Trade** connects **Profile** (seller and buyer) with **Pokemon**.
* **Market** links Pokémon with pricing and listing details.

Implementation Steps

**Set Up Django Project**:

* Create a new Django project and app.

django-admin startproject poketrade

cd poketrade

django-admin startapp trades

**Define Models**:

* Create the models as described above in models.py of your app.

**Migrate the Database**:

Run migrations to create the database tables.

python manage.py makemigrations

python manage.py migrate

**Admin Interface**:

* Register your models in admin.py to manage them easily via the Django admin panel.

from django.contrib import admin

from .models import Profile, Pokemon, Trade, Market

admin.site.register(Profile)

admin.site.register(Pokemon)

admin.site.register(Trade)

admin.site.register(Market)

**Create Views and Templates**:

* Develop views for user registration, Pokémon listing, trade offers, and trading actions. Use Django’s built-in views for user authentication and extend them as needed.

 **APIs (Optional)**:

* If you want to create a more dynamic user experience, consider building RESTful APIs using Django REST Framework to handle trading and Pokémon management.

 **Testing and Deployment**:

* Test your application thoroughly, especially the trading logic, to prevent any exploits.
* Deploy using platforms like Heroku or DigitalOcean once you’re satisfied with the functionality.

**Additional Considerations**

* **Caching**: Implement caching for frequently accessed data to improve performance.
* **Security**: Ensure that user authentication and data protection practices are in place.
* **Scalability**: Plan your database structure to allow for potential future expansions (e.g., adding new Pokémon generations).

**Creating Views**

**Django views are responsible for processing requests and returning responses. You’ll want to create several views to handle user interactions.**

**1. User Registration and Profile View**

**from django.shortcuts import render, redirect**

**from django.contrib.auth import login, authenticate**

**from django.contrib.auth.forms import UserCreationForm**

**from .models import Profile**

**def register(request):**

**if request.method == 'POST':**

**form = UserCreationForm(request.POST)**

**if form.is\_valid():**

**user = form.save()**

**Profile.objects.create(user=user)**

**login(request, user)**

**return redirect('home')**

**else:**

**form = UserCreationForm()**

**return render(request, 'register.html', {'form': form})**

1. **Pokémon Listing View**

**from django.shortcuts import render**

**from .models import Pokemon**

**def pokemon\_list(request):**

**pokemons = Pokemon.objects.all()**

**return render(request, 'pokemon\_list.html', {'pokemons': pokemons})**

1. **Trading View**

**from django.shortcuts import get\_object\_or\_404**

**from .models import Trade, Pokemon**

**def trade\_pokemon(request, pokemon\_id):**

**pokemon = get\_object\_or\_404(Pokemon, id=pokemon\_id)**

**if request.method == 'POST':**

**# Implement trading logic here**

**trade = Trade(seller=request.user.profile, pokemon=pokemon)**

**trade.save()**

**return redirect('pokemon\_list')**

**return render(request, 'trade\_pokemon.html', {'pokemon': pokemon})**

**Creating Templates**

**Django templates allow you to create dynamic HTML pages. Here are examples for the views:**

* 1. **Base Template (base.html)**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>PokéTrade Quest</title>**

**</head>**

**<body>**

**<header>**

**<h1>PokéTrade Quest</h1>**

**<nav>**

**<a href="{% url 'pokemon\_list' %}">Pokémon</a>**

**<a href="{% url 'register' %}">Register</a>**

**<a href="{% url 'logout' %}">Logout</a>**

**</nav>**

**</header>**

**<main>**

**{% block content %}**

**{% endblock %}**

**</main>**

**</body>**

**</html>**

* 1. **Register Template (register.html)**

**{% extends 'base.html' %}**

**{% block content %}**

**<h2>Register</h2>**

**<form method="post">**

**{% csrf\_token %}**

**{{ form.as\_p }}**

**<button type="submit">Register</button>**

**</form>**

**{% endblock %}**

* 1. **Pokémon List Template (pokemon\_list.html)**

**{% extends 'base.html' %}**

**{% block content %}**

**<h2>Available Pokémon</h2>**

**<ul>**

**{% for pokemon in pokemons %}**

**<li>**

**{{ pokemon.name }} (Level {{ pokemon.level }})**

**<a href="{% url 'trade\_pokemon' pokemon.id %}">Trade</a>**

**</li>**

**{% endfor %}**

**</ul>**

**{% endblock %}**

**Creating APIs**

**To create RESTful APIs, you can use Django REST Framework (DRF). Here’s how you can set it up:**

**1. Install Django REST Framework**

**pip install djangorestframework**

**2. Add DRF to Installed Apps**

**In settings.py:**

**INSTALLED\_APPS = [**

**...**

**'rest\_framework',**

**]**

**3. Create Serializers**

**Serializers convert complex data types like querysets into JSON.**

**from rest\_framework import serializers**

**from .models import Pokemon, Profile, Trade**

**class PokemonSerializer(serializers.ModelSerializer):**

**class Meta:**

**model = Pokemon**

**fields = '\_\_all\_\_'**

**class ProfileSerializer(serializers.ModelSerializer):**

**class Meta:**

**model = Profile**

**fields = ['user', 'experience', 'total\_trades']**

**4. Create API Views**

**Use DRF views to handle requests.**

**from rest\_framework import viewsets**

**from .models import Pokemon**

**from .serializers import PokemonSerializer**

**class PokemonViewSet(viewsets.ModelViewSet):**

**queryset = Pokemon.objects.all()**

**serializer\_class = PokemonSerializer**

**5. Set Up URLs for the API**

**In your app's urls.py:**

**from django.urls import path, include**

**from rest\_framework.routers import DefaultRouter**

**from .views import PokemonViewSet**

**router = DefaultRouter()**

**router.register(r'pokemons', PokemonViewSet)**

**urlpatterns = [**

**path('api/', include(router.urls)),**

**]**

**Testing Your APIs**

**You can test your APIs using tools like Postman or curl. For example, to get a list of all Pokémon:**

**GET /api/pokemons/**

**Summary**

* **Views handle the business logic of your application.**
* **Templates are used for rendering HTML pages.**
* **APIs allow for a more dynamic user experience and can enable third-party integrations.**

**Updated trade\_pokemon View with Trading Logic**

**Here’s how you can implement the trading logic step-by-step:**

1. **Check if the Pokémon is tradeable.**
2. **Handle trade acceptance from the buyer.**
3. **Update the owners of the Pokémon.**
4. **Update experience points.**
5. **Mark the trade as completed.**

**from django.shortcuts import get\_object\_or\_404, redirect**

**from django.contrib import messages**

**from .models import Trade, Pokemon**

**def trade\_pokemon(request, pokemon\_id):**

**pokemon = get\_object\_or\_404(Pokemon, id=pokemon\_id)**

**if request.method == 'POST':**

**# Check if the Pokémon is available for trade**

**if not pokemon.is\_tradeable:**

**messages.error(request, "This Pokémon is not available for trade.")**

**return redirect('pokemon\_list')**

**# If the user is not the owner of the Pokémon**

**if pokemon.owner == request.user.profile:**

**messages.error(request, "You cannot trade your own Pokémon.")**

**return redirect('pokemon\_list')**

**# Create a trade record**

**trade = Trade(seller=pokemon.owner, buyer=request.user.profile, pokemon=pokemon)**

**trade.save()**

**# Update the Pokémon ownership**

**pokemon.owner = request.user.profile**

**pokemon.is\_tradeable = False # Mark it as no longer available for trade**

**pokemon.save()**

**# Update experience points**

**request.user.profile.experience += 10 # Example XP for trading**

**request.user.profile.total\_trades += 1**

**request.user.profile.save()**

**# Show a success message**

**messages.success(request, f"You successfully traded for {pokemon.name}!")**

**return redirect('pokemon\_list')**

**return render(request, 'trade\_pokemon.html', {'pokemon': pokemon})**

**Explanation of the Logic**

1. **Check if the Pokémon is tradeable:**
   * **The is\_tradeable field prevents trading of Pokémon that are not currently available.**
2. **Ownership Validation:**
   * **The view checks if the current user is trying to trade their own Pokémon. If so, an error message is displayed.**
3. **Creating a Trade Record:**
   * **A new Trade instance is created, linking the seller (previous owner) and buyer (current user) with the Pokémon.**
4. **Updating Pokémon Ownership:**
   * **The owner field of the Pokemon model is updated to reflect the new owner.**
   * **The Pokémon is marked as not tradeable after the trade.**
5. **Experience Points Update:**
   * **The user’s experience points and total trades are updated to reflect the completed trade.**
6. **User Feedback:**
   * **Success or error messages are displayed to the user using Django’s messages framework.**

**Additional Considerations**

* **Trade Cancellation: You might want to implement a way to cancel trades if they haven’t been completed.**
* **Trade Offers: Consider allowing users to make offers for Pokémon rather than immediate trades, adding complexity to the trading system.**
* **Validation and Edge Cases: Ensure proper validation for edge cases, such as attempting to trade with insufficient Pokémon or invalid actions.**

**To implement experience tracking for Pokémon and allow them to gain experience from trades, we’ll need to update the Pokemon model to include an experience field and add logic to handle leveling up.**

**Updated Pokémon Model**

**Here’s how you can modify the Pokemon model to track current experience and implement the leveling logic:**

**from django.db import models**

**class Pokemon(models.Model):**

**name = models.CharField(max\_length=50)**

**level = models.IntegerField(default=1)**

**current\_experience = models.IntegerField(default=0) # New field to track current experience**

**owner = models.ForeignKey('Profile', related\_name='pokemons', on\_delete=models.CASCADE)**

**is\_tradeable = models.BooleanField(default=True)**

**def add\_experience(self, amount):**

**"""Add experience and level up if needed."""**

**self.current\_experience += amount**

**# Check if the Pokémon levels up**

**while self.current\_experience >= 100:**

**self.current\_experience -= 100**

**self.level += 1**

**# Prevent leveling above level 100**

**if self.level >= 100:**

**self.level = 100**

**self.current\_experience = 0 # Reset experience at max level**

**break**

**self.save()**

**Explanation of the Updated Model**

* **current\_experience: This field tracks the Pokémon's current experience points.**
* **add\_experience method: This method adds experience points to the Pokémon. If the total experience reaches or exceeds 100, it levels up the Pokémon. The leveling up continues until the Pokémon's experience is less than 100.**

**Updating the Trade Logic**

**Now, let’s modify the trade\_pokemon view to grant experience points to the traded Pokémon. When a Pokémon is traded, it gains a certain amount of experience.**

**from django.shortcuts import get\_object\_or\_404, redirect**

**from django.contrib import messages**

**from .models import Trade, Pokemon**

**def trade\_pokemon(request, pokemon\_id):**

**pokemon = get\_object\_or\_404(Pokemon, id=pokemon\_id)**

**if request.method == 'POST':**

**# Check if the Pokémon is available for trade**

**if not pokemon.is\_tradeable:**

**messages.error(request, "This Pokémon is not available for trade.")**

**return redirect('pokemon\_list')**

**# If the user is not the owner of the Pokémon**

**if pokemon.owner == request.user.profile:**

**messages.error(request, "You cannot trade your own Pokémon.")**

**return redirect('pokemon\_list')**

**# Create a trade record**

**trade = Trade(seller=pokemon.owner, buyer=request.user.profile, pokemon=pokemon)**

**trade.save()**

**# Update the Pokémon ownership**

**old\_owner = pokemon.owner**

**pokemon.owner = request.user.profile**

**pokemon.is\_tradeable = False # Mark it as no longer available for trade**

**# Grant experience for trading**

**experience\_gain = 20 # Example XP gained from the trade**

**pokemon.add\_experience(experience\_gain)**

**# Update old owner's experience as well if necessary**

**old\_owner.experience += 5 # Example XP for the seller**

**old\_owner.save()**

**# Show a success message**

**messages.success(request, f"You successfully traded for {pokemon.name}!")**

**return redirect('pokemon\_list')**

**return render(request, 'trade\_pokemon.html', {'pokemon': pokemon})**

**Key Changes in the Trade Logic**

* **Experience Gain: When a Pokémon is traded, it receives a specified amount of experience (e.g., 20 XP). This is done using the add\_experience method.**
* **Old Owner Experience: Optionally, you can give the previous owner a small amount of experience (e.g., 5 XP) as a reward for the trade.**

**Additional Considerations**

* **Experience Gain Logic: You can adjust the experience points awarded based on various factors (e.g., rarity of Pokémon, level difference).**
* **Display Experience and Level: Update your templates to display the current level and experience of each Pokémon.**

**To implement the display of a Pokémon’s experience and level in your templates and allow users to pass their experience gain from trades to a Pokémon, follow these steps:**

**Step 1: Update Templates to Display Level and Experience**

**Pokémon List Template (pokemon\_list.html)**

**Modify the Pokémon list to show each Pokémon's level and current experience:**

**{% extends 'base.html' %}**

**{% block content %}**

**<h2>Available Pokémon</h2>**

**<ul>**

**{% for pokemon in pokemons %}**

**<li>**

**{{ pokemon.name }} (Level {{ pokemon.level }}, XP {{ pokemon.current\_experience }})**

**<a href="{% url 'trade\_pokemon' pokemon.id %}">Trade</a>**

**<form method="post" action="{% url 'pass\_experience' pokemon.id %}">**

**{% csrf\_token %}**

**<button type="submit">Pass Experience</button>**

**</form>**

**</li>**

**{% endfor %}**

**</ul>**

**{% endblock %}**

**Step 2: Implement Experience Transfer Logic**

**Create a View for Passing Experience**

**Add a view to handle passing experience from the user to a specific Pokémon. This view will deduct a specified amount of experience from the user's profile and add it to the Pokémon.**

**from django.shortcuts import get\_object\_or\_404, redirect**

**from django.contrib import messages**

**from .models import Pokemon, Profile**

**def pass\_experience(request, pokemon\_id):**

**pokemon = get\_object\_or\_404(Pokemon, id=pokemon\_id)**

**if request.method == 'POST':**

**# Example: Assume user can pass 10 XP at a time**

**experience\_to\_pass = 10**

**# Check if user has enough experience to pass**

**if request.user.profile.experience < experience\_to\_pass:**

**messages.error(request, "You do not have enough experience to pass.")**

**return redirect('pokemon\_list')**

**# Deduct experience from the user**

**request.user.profile.experience -= experience\_to\_pass**

**request.user.profile.save()**

**# Pass experience to the Pokémon**

**pokemon.add\_experience(experience\_to\_pass)**

**# Show success message**

**messages.success(request, f"You passed {experience\_to\_pass} XP to {pokemon.name}!")**

**return redirect('pokemon\_list')**

**return redirect('pokemon\_list')**

**Step 3: Update URLs**

**Add a URL pattern for the new experience transfer view in your app’s urls.py:**

**from django.urls import path**

**from .views import trade\_pokemon, pass\_experience**

**urlpatterns = [**

**path('trade/<int:pokemon\_id>/', trade\_pokemon, name='trade\_pokemon'),**

**path('pass\_experience/<int:pokemon\_id>/', pass\_experience, name='pass\_experience'),**

**]**

**Step 4: Update User Profile to Display Experience**

**Ensure the user profile page (if you have one) displays the user's total experience as well, so users can track their points.**

**User Profile Template (e.g., profile.html)**

**<h2>{{ user.username }}'s Profile</h2>**

**<p>Total Experience: {{ user.profile.experience }}</p>**

**<ul>**

**{% for pokemon in user.profile.pokemons.all %}**

**<li>{{ pokemon.name }} (Level {{ pokemon.level }}, XP {{ pokemon.current\_experience }})</li>**

**{% endfor %}**

**</ul>**

**Summary of Changes**

1. **Pokémon List Display: The Pokémon list now shows the level and current experience of each Pokémon.**
2. **Pass Experience Button: Users can pass a fixed amount of experience to their Pokémon directly from the Pokémon list.**
3. **Experience Transfer Logic: A new view handles the logic for passing experience and checks if the user has enough experience to do so.**