1. Difference between Django REST Framework (DRF) and FastAPI

Key Features:

- Django REST Framework (DRF):

  DRF is built on top of Django, which is a popular and feature-packed web framework. It offers a lot of built-in tools to make creating RESTful APIs easier, like serializers, viewsets, and authentication mechanisms. Since it integrates with Django's ORM, managing the database becomes pretty straightforward. DRF is a great choice when you need a full-stack solution, especially if you're building something complex with user authentication, admin panels, or intricate data models. That said, it works synchronously, so when you have a lot of simultaneous requests, things can slow down a bit.

- FastAPI:

  FastAPI is all about speed and performance, especially when dealing with lots of requests. It's designed to take full advantage of asynchronous programming, meaning it can handle many requests at once without blocking. One of its coolest features is automatic documentation generation using OpenAPI, which makes it super easy to understand the API. FastAPI also uses Python's type hints for validation, which makes your code cleaner and helps catch errors early. It's a perfect fit for modern applications where you need things to run fast and scale well.

Advantages:

- DRF:

  DRF is perfect if you want a mature, all-in-one solution with tons of built-in tools. It's great for large apps where you need things like user management, authentication, and admin dashboards. Plus, Django's ORM makes working with databases easier. It's definitely not the fastest option out there, but for full-stack apps that need a lot of features, DRF has got you covered.

- FastAPI:

  FastAPI's main selling point is speed. Since it's asynchronous, it can handle a lot of requests at once, making it super fast. It's also great for building APIs that need to scale, like real-time apps or microservices. Plus, the automatic OpenAPI docs are a huge win during development, and using type hints makes your code easier to work with. If you need an API that's lightning-fast and efficient, FastAPI is hard to beat.

Disadvantages:

- DRF:

  While DRF is powerful, it can be slower than FastAPI, especially when you have high traffic. Its synchronous nature means that requests are handled one by one, which can cause delays in busy apps. Additionally, Django can feel a bit heavy if you're only building simple APIs or microservices. The learning curve can also be a bit steep if you're not familiar with Django's ecosystem.

- FastAPI:

  FastAPI is newer and doesn't have the same long history as Django, so its ecosystem is a little smaller. You may need extra setup for things like authentication, which Django makes super easy. Also, it doesn't come with a built-in admin interface like Django, so if you need something like that, you'll have to set it up yourself.

2. Challenges and Solutions

Django REST Framework (DRF):

- Challenge 1: Slow response times in large-scale applications

  Since DRF works synchronously, it can struggle with handling lots of requests at once, which can lead to slower performance when the app gets heavy traffic. Each request is processed one after the other, which can create bottlenecks.

  Solution: I tried optimizing database queries and switching to async views, but it's hard to completely solve performance issues with DRF in high-traffic situations. In the end, breaking the app into microservices helped with scaling, but the bottleneck issue remained.

- Challenge 2: Difficulty in scaling

  Scaling Django-based apps can be tough because it's a monolithic framework. As you add more users and services, managing everything can get complicated.

  Solution: I relied on caching and optimized database queries to improve performance, but when things got larger, it made more sense to break the app up into smaller, more manageable microservices.

- Challenge 3: Complex deployment configurations

  DRF apps often require a bunch of additional services like Celery for background tasks, which can make deployment a bit more complicated.

  Solution: I turned to Docker and containerization, which simplified things, but the overall setup still felt a bit more complicated than it needed to be.

FastAPI:

- Challenge 1: Learning curve for async features

  Since FastAPI uses async/await from the start, it took a little while to get comfortable with asynchronous programming.

  Solution: I focused on learning how to leverage FastAPI's async capabilities to improve performance, especially for handling a high number of concurrent requests. The async features ended up being a huge advantage for scalability.

- Challenge 2: Database session management

  Managing database sessions in an async environment was a bit tricky at first, and I had to be careful to ensure sessions were properly closed after each request.

  Solution: FastAPI's dependency injection system helped a lot here, as it allowed me to handle database sessions more efficiently without worrying about session leaks.

- Challenge 3: Compatibility with frontend

  CORS (Cross-Origin Resource Sharing) issues popped up when I was connecting FastAPI with the React frontend, especially during local development.

  Solution: I used FastAPI's built-in CORS middleware to allow my frontend and backend to communicate smoothly, making sure to adjust it properly for production environments.

1. Project Overview

- A FastAPI backend for a To-Do List
- Integrated with a React frontend
- PostgreSQL used as the database (hosted on Render)

2. Differences Between DRF and FastAPI

- DRF: Heavier, more batteries-included, great for complex auth & admin
- FastAPI: Lightweight, async by default, faster for APIs

3. Challenges & Solutions

| Challenge | Solution |
| --- | --- |
| CORS errors | Used `CORSMiddleware` in FastAPI |
| Auth mismatch | Switched to unauthenticated public API |
| Render build configs | Set custom start/build commands |

4. Deployment Links

- Backend: https://backend-fastapi-obja.onrender.com
- Frontend: https://phenomenal-cheesecake-fc3ec5.netlify.app