# Report: Comparison of DRF and FastAPI for To-Do List Application

## Difference Between Django REST Framework (DRF) and FastAPI

**Django REST Framework (DRF)** and **FastAPI** are both popular frameworks for building RESTful APIs in Python. Written below are their differences, advantages, and disadvantages, tailored for aTo-Do List application context.

### Django REST Framework (DRF)

- **Key Features**:
  - Built on Django, leveraging its ORM for database interactions.
  - Provides robust serializers for data validation and transformation (e.g., converting TodoItem models to JSON).
  - Includes class-based views, authentication, and permissions readily made.
  - Admin interface for user friendly management.
- **Advantages**:
  - Stable, with extensive community support.
  - Ideal for complex projects needing integrated features like a full admin dashboard for management.
  - Strong ORM simplifies database queries and making database more secured.
- **Disadvantages**:
  - Heavier and slower compared to lightweight frameworks that could impact performance for simple APIs.
  - It would take quite a longer time to learn due to its complexity
  - Overkill for minimal APIs like a To-Do list application, where only CRUD operations were needed.

### FastAPI

- **Key Features**:
  - High performance focus framework.
  - Automatic OpenAPI (Swagger) documentation (e.g., /docs for testing /todos endpoints).
  - Type hints and Pydantic models for robust data validation (e.g., TodoCreate, TodoUpdate).
  - Dependency injection for flexible request handling (e.g., get_db for SQLAlchemy sessions).
  - Lightweight and focused on APIs, not full web apps.
- **Advantages**:
  - Exceptional performance due to async support, ideal for real-time or high-traffic APIs.

    - Simpler setup for small projects like our To-Do List (e.g., quick CRUD endpoints).
    - Auto-generated Swagger UI simplifies testing.
- **Disadvantages**:
    - Less built-in functionality compared to Django (e.g., no admin panel).
    - Can be complex for beginners.
    - Requires external libraries for advanced features (e.g., authentication).
    - Limited ORM integration, relying on SQLAlchemy for our PostgreSQL setup.

## Challenges in Development and Deployment

### Django REST Framework

- **Development Challenge**: **Complex Configuration**
    - **Issue**: Setting up DRF requires configuring Django's settings (e.g., INSTALLED_APPS, REST_FRAMEWORK), models, serializers, and views, which is a lot of work for a simple To-Do List app.
    - **Solution**: Use Django's startproject and startapp to scaffold, and leverage generic views (e.g., ListCreateAPIView.
    -
- **Deployment Challenge**: **Performance Optimization**
    - **Issue**: Django's heavyweight stack can lead to slower response times on basic hosting like Render especially for frequent GET requests.
    - **Solution**: Deploy with Gunicorn and a WSGI server, and optimize database queries.

### FastAPI Challenges

- **Development Challenge**: **Filter Endpoint Integration**
    - **Issue**: The frontend's "Completed" filter (GET /todos/filter?completed=true) failed, showing "Failed to fetch tasks," suggesting the issue could be the backend's filter endpoint handling the completed request or database query.
    - **Solution**: Revised the /todos/filter endpoint to use Optional[bool] for the completed parameter and added SQLAlchemy filter. (db.query(models.TodoItem).filter(models.TodoItem.completed == completed)). This will ensure that the response matches with the frontend App.js expected format ([{id: int, title: str, completed: bool}, ...]).
- **Development Challenge**: **CORS Configuration**
    - **Issue**: Initial frontend-backend communication failed due to CORS restrictions, preventing locally hosted react app from accessing FastAPI (http://localhost:8000).
    - **Solution**: Added CORSMiddleware with allow_origins=["*"] in main.py to allow all origins during development.

- **Deployment Challenge**: **Database Connectivity**
    - **Issue**: Connecting to the Render-hosted PostgreSQL database requires a correct DATABASE_URL.
    - **Solution**: Used the provided DATABASE_URL (postgresql://todo_backend_postgres_user:...) in database.py, configured SQLAlchemy's create_engine, and tested connectivity locally before deployment.
- **Deployment Challenge**: **Render Deployment Setup**
    - **Issue**: Deploying FastAPI to Render required proper build and start commands to run Uvicorn and handle environment variables.
    - **Solution**: Set the build command to "pip install -r requirements.txt" and start command to "uvicorn main:app --host 0.0.0.0 --port 8000".

## Conclusion

FastAPI is ideal for a To-Do List app with its simplicity, performance, and automatic documentation, making development and testing straightforward. Django in the other hand, while powerful, would have added unnecessary complexity for this scope. Challenges like filter endpoint issues and CORS were resolved by refining endpoint logic and middleware, while deployment to Render was streamlined with clear configurations. This experience highlights FastAPI's edge for lightweight APIs and DRF's strength for feature-rich applications.