Logs Dashboard — Full-Stack Step-by-Step Guide

Tech stack - Frontend: React + TypeScript (Vite) + Tailwind CSS - Charts: Recharts - Backend: Django + Django REST Framework - DB: MySQL (or use MariaDB) - Dev tooling: Docker Compose (optional) or local virtualenv + node

---

What you'll find in this document

1. Project overview & file structure
2. Backend: full step-by-step (Django app, models, serializers, viewsets, filters, aggregated endpoints)
3. Frontend: full step-by-step (Vite React TypeScript + Tailwind setup, API client, pages, chart)
4. Docker / docker-compose to run everything locally
5. Extras: CSV export, tests, sample data, README suggestions

---

1. Project layout (recommended)

```
logs-dashboard/
├─ backend/
│  ├─ Dockerfile
│  ├─ requirements.txt
│  ├─ manage.py
│  ├─ logs_project/
│  │  ├─ settings.py
│  │  ├─ urls.py
│  │  └─ wsgi.py
│  └─ logs_app/
│     ├─ models.py
│     ├─ serializers.py
│     ├─ views.py
│     ├─ filters.py
│     ├─ urls.py
│     ├─ admin.py
│     └─ migrations/
├─ frontend/
│  ├─ Dockerfile
│  ├─ package.json
│  └─ src/
│     ├─ main.tsx
│     ├─ App.tsx
│     ├─ api/
│     │  └─ api.ts
```

```
│     ├─ pages/
│     │  ├─ LogList.tsx
│     │  ├─ LogDetail.tsx
│     │  ├─ CreateLog.tsx
│     │  └─ Dashboard.tsx
│     ├─ components/
│     │  ├─ FilterPanel.tsx
│     │  ├─ LogTable.tsx
│     │  └─ TrendChart.tsx
│     └─ styles/tailwind.css
└─ docker-compose.yml
```

---

2. Backend (Django) — step by step

2.0 Requirements (examples)

`requirements.txt`

```
Django>=4.2
djangorestframework
django-filter
django-cors-headers
pymysql
python-dateutil
```

> I recommend **pymysql** because it avoids compiling `mysqlclient` while developing locally. For production you may switch to `mysqlclient`.

2.1 Create project & app

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
django-admin startproject logs_project .
python manage.py startapp logs_app
```

2.2 settings.py (important parts)

- Add installed apps: `rest_framework`, `django_filters`, `corsheaders`, `logs_app`
- DB settings (MySQL) example using env vars
- Configure CORS for frontend dev

```
# settings.py (snippets)
INSTALLED_APPS = [
    # ...
    'corsheaders',
    'rest_framework',
    'django_filters',
    'logs_app',
]
MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
    # ...
]
CORS_ORIGIN_ALLOW_ALL = True  # for dev only; prefer CORS_ORIGIN_WHITELIST in
prod

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': os.environ.get('MYSQL_DATABASE', 'logsdb'),
        'USER': os.environ.get('MYSQL_USER', 'root'),
        'PASSWORD': os.environ.get('MYSQL_PASSWORD', ''),
        'HOST': os.environ.get('MYSQL_HOST', 'db'),
        'PORT': os.environ.get('MYSQL_PORT', '3306'),
        'OPTIONS': {
            'init_command': "SET sql_mode='STRICT_TRANS_TABLES'",
        }
    }
}

REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': [
        'django_filters.rest_framework.DjangoFilterBackend',
        'rest_framework.filters.SearchFilter',
        'rest_framework.filters.OrderingFilter',
    ],
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 20,
}
```

If you use `pymysql`, add at top of `manage.py` or `logs_project/__init__.py`:

```
import pymysql
pymysql.install_as_MySQLdb()
```

2.3 Model (logs_app/models.py)

```
from django.db import models
from django.utils import timezone

class Log(models.Model):
    DEBUG = 'DEBUG'
    INFO = 'INFO'
    WARNING = 'WARNING'
    ERROR = 'ERROR'
    CRITICAL = 'CRITICAL'

    SEVERITY_CHOICES = [
        (DEBUG, 'Debug'),
        (INFO, 'Info'),
        (WARNING, 'Warning'),
        (ERROR, 'Error'),
        (CRITICAL, 'Critical'),
    ]

    timestamp = models.DateTimeField(default=timezone.now, db_index=True)
    message = models.TextField()
    severity = models.CharField(max_length=10, choices=SEVERITY_CHOICES,
db_index=True)
    source = models.CharField(max_length=100, db_index=True)

    class Meta:
        ordering = ['-timestamp']

    def __str__(self):
        return f"[{self.timestamp}] {self.severity} - {self.source}"
```

Add indexes if you expect large volume.

2.4 Serializers (logs_app/serializers.py)

```
from rest_framework import serializers
from .models import Log

class LogSerializer(serializers.ModelSerializer):
    class Meta:
        model = Log
        fields = ['id', 'timestamp', 'message', 'severity', 'source']
```

2.5 Filters (logs_app/filters.py)

```
import django_filters
from .models import Log

class LogFilter(django_filters.FilterSet):
    date_from = django_filters.IsoDateTimeFilter(field_name='timestamp',
lookup_expr='gte')
    date_to = django_filters.IsoDateTimeFilter(field_name='timestamp',
lookup_expr='lte')

    class Meta:
        model = Log
        fields = ['severity', 'source', 'date_from', 'date_to']
```

2.6 Views (logs_app/views.py)

```
from rest_framework import viewsets, status
from rest_framework.decorators import action
from rest_framework.response import Response
from django.db.models import Count
from django.db.models.functions import TruncDay
from .models import Log
from .serializers import LogSerializer
from .filters import LogFilter

class LogViewSet(viewsets.ModelViewSet):
    queryset = Log.objects.all()
    serializer_class = LogSerializer
    filterset_class = LogFilter
    search_fields = ['message', 'source']
    ordering_fields = ['timestamp', 'severity', 'source']

    @action(detail=False, methods=['get'])
    def raw(self, request):
        """
        Returns raw logs (no pagination) matching filters
        query params: date_from, date_to, severity, source
        """
        qs = self.filter_queryset(self.get_queryset())
        serializer = self.get_serializer(qs, many=True)
        return Response(serializer.data)

    @action(detail=False, methods=['get'])
    def aggregated(self, request):
        """
```

```
        Returns aggregated data. Params:
        - group_by: date|severity|source (default: date)
        - date_from, date_to, severity, source
        - interval: day|month (only when group_by=date)
        """
        group_by = request.query_params.get('group_by', 'date')
        interval = request.query_params.get('interval', 'day')
        qs = self.filter_queryset(self.get_queryset())

        if group_by == 'date':
            # truncate to day (or month if needed)
            if interval == 'day':
                ts = TruncDay('timestamp')
            else:
                # truncate month
                from django.db.models.functions import TruncMonth
                ts = TruncMonth('timestamp')

            data =
qs.annotate(period=ts).values('period').annotate(count=Count('id')).order_by('period')
            # normalize dates to ISO strings
            result = [{'date': item['period'].date().isoformat(), 'count':
item['count']} for item in data]
            return Response(result)

        elif group_by == 'severity':
            data = qs.values('severity').annotate(count=Count('id')).order_by('-
count')
            return Response(list(data))

        else:  # source
            data = qs.values('source').annotate(count=Count('id')).order_by('-
count')
            return Response(list(data))
```

2.7 URLs (logs_app/urls.py)

```
from rest_framework.routers import DefaultRouter
from .views import LogViewSet

router = DefaultRouter()
router.register(r'logs', LogViewSet, basename='log')

urlpatterns = router.urls
```

Then include in project `urls.py`:

```
from django.urls import path, include

urlpatterns = [
    path('api/', include('logs_app.urls')),
]
```

2.8 Migrations, admin, sample data

```
python manage.py makemigrations
python manage.py migrate
python manage.py createsuperuser  # optional
```

In `admin.py` register the `Log` model for quick management.

To add sample data, you can create a management command or a small script that uses Django ORM to bulk_create logs.

2.9 Testing the API quickly

Use `curl` or Postman. Examples: - List logs paginated: `GET /api/logs/` - Get raw logs: `GET /api/logs/raw/?date_from=2025-09-01T00:00:00Z&date_to=2025-09-30T23:59:59Z` - Aggregated by date: `GET /api/logs/aggregated/?date_from=2025-09-01&date_to=2025-09-30&group_by=date&interval=day`

---

3. Frontend — React TypeScript + Tailwind (step by step)

3.0 Create project (Vite)

```
# from repo root
cd frontend
npm create vite@latest . -- --template react-ts
npm install
```

3.1 Install packages

```
npm install axios react-router-dom@6 recharts dayjs papaparse
# dev
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

`tailwind.config.cjs` sample:

```
module.exports = {
  content: ['./index.html', './src/**/*.{ts,tsx}'],
  theme: { extend: {} },
  plugins: [],
}
```

`src/styles/tailwind.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Import that file into `main.tsx`.

3.2 API client (src/api/api.ts)

```
import axios from 'axios'

const API_URL = import.meta.env.VITE_API_URL || 'http://localhost:8000/api'

export const api = axios.create({
  baseURL: API_URL,
  timeout: 10000,
})

export type Log = {
  id: number
  timestamp: string
  message: string
  severity: 'DEBUG'|'INFO'|'WARNING'|'ERROR'|'CRITICAL'
  source: string
}
```

3.3 Routing (src/App.tsx)

```
import { BrowserRouter, Routes, Route } from 'react-router-dom'
import LogList from './pages/LogList'
import LogDetail from './pages/LogDetail'
import CreateLog from './pages/CreateLog'
import Dashboard from './pages/Dashboard'
```

```
export default function App(){
  return (
    <BrowserRouter>
      <Routes>
        <Route path='/' element={<Dashboard/>} />
        <Route path='/logs' element={<LogList/>} />
        <Route path='/logs/new' element={<CreateLog/>} />
        <Route path='/logs/:id' element={<LogDetail/>} />
      </Routes>
    </BrowserRouter>
  )
}
```

3.4 LogList (src/pages/LogList.tsx) — key features

- search, filter (severity, source), sort, pagination
- uses `api.get('/logs/')` with query params

Key idea (pseudo):

```
// simplified version
const [logs, setLogs] = useState<Log[]>([])
const [page, setPage] = useState(1)
const [q, setQ] = useState('')

useEffect(()=>{
  api.get('/logs/', { params: { page, search: q, severity, source,
ordering } }).then(r=> setLogs(r.data.results))
}, [page,q,severity,source,ordering])
```

Render table with Tailwind classes; make rows clickable to go to detail page.

3.5 LogDetail (src/pages/LogDetail.tsx)

- Fetch single log by id `GET /api/logs/{id}/`
- Show editable form (timestamp, message, severity, source)
- Submit `PUT /api/logs/{id}/` to update
- Delete with `DELETE /api/logs/{id}/`

3.6 CreateLog

- Simple form POST to `/api/logs/` then redirect to list

3.7 Dashboard & Chart (src/pages/Dashboard.tsx)

- Filter panel: date range input (use simple two `<input type='date'/>` or a date picker lib)

- When filters change, call `/api/logs/aggregated/?group_by=date&date_from=...&date_to=...` to fetch counts by day
- Render `TrendChart` that accepts data array `[{date: '2025-09-01', count: 12}, ...]`

`TrendChart.tsx` (Recharts)

```
import { LineChart, Line, XAxis, YAxis, CartesianGrid, Tooltip, Legend,
ResponsiveContainer } from 'recharts'

export default function TrendChart({data}:{data:{date:string,count:number}[]}){
  return (
    <ResponsiveContainer width='100%' height={320}>
      <LineChart data={data}>
        <CartesianGrid strokeDasharray='3 3' />
        <XAxis dataKey='date' />
        <YAxis />
        <Tooltip />
        <Legend />
        <Line type='monotone' dataKey='count' name='Log Count' />
      </LineChart>
    </ResponsiveContainer>
  )
}
```

3.8 CSV Export

Use `papaparse` or build CSV string manually:

```
import { saveAs } from 'file-saver'
import Papa from 'papaparse'

const exportCsv = (rows:any[]) => {
  const csv = Papa.unparse(rows)
  const blob = new Blob([csv], { type: 'text/csv;charset=utf-8;' })
  saveAs(blob, 'logs_export.csv')
}
```

3.9 UX notes / Tailwind

- Use responsive table: `overflow-x-auto` wrapper
- Use `p-4`, `rounded-xl`, `shadow` for cards
- Keep filter panel collapsible on mobile

---

4. Docker Compose (optional, quick local run)

`docker-compose.yml` (simplified)

```
version: '3.8'
services:
  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: rootpassword
      MYSQL_DATABASE: logsdb
      MYSQL_USER: logsuser
      MYSQL_PASSWORD: logspass
    ports:
      - '3306:3306'
    volumes:
      - db_data:/var/lib/mysql

  web:
    build: ./backend
    command: bash -c "python manage.py migrate && python manage.py runserver
0.0.0.0:8000"
    volumes:
      - ./backend:/app
    ports:
      - '8000:8000'
    depends_on:
      - db
    environment:
      MYSQL_HOST: db
      MYSQL_DATABASE: logsdb
      MYSQL_USER: logsuser
      MYSQL_PASSWORD: logspass

  frontend:
    build: ./frontend
    command: sh -c "npm install && npm run dev -- --host 0.0.0.0"
    ports:
      - '5173:5173'
    volumes:
      - ./frontend:/app
    environment:
      VITE_API_URL: 'http://host.docker.internal:8000/api'
    depends_on:
      - web
```

```
volumes:
  db_data:
```

> Note: Using `host.docker.internal` works on Docker Desktop for mac/windows. For Linux use network mode or set frontend API base to `http://web:8000/api` and ensure CORS allows it.

---

5. Tests and quality

- Backend: use Django tests (`tests.py`) or Pytest-Django. Write unit tests for viewset filters and aggregated endpoint.
- Frontend: use React Testing Library + Vitest or Jest for component tests (snapshot for chart, behavior for list pagination).

Example backend test idea: - Create many logs across dates and severities, call `/api/logs/aggregated/?group_by=severity` and assert counts per severity.

---

6. Bonus ideas (pick any)

- Real-time log streaming via WebSocket (Django Channels) and live chart updates.
- File upload endpoint to ingest large log files (CSV) and background processing (Celery + Redis).
- Export filtered logs to CSV from backend (streaming response for big datasets).
- Histogram / donut chart showing severity distribution.
- Authentication (JWT) and per-user saved filters.

---

7. README notes to include in submission

- How to run (both docker and local)
- Endpoints list with examples
- Design decisions (why DRF, why Recharts, why MySQL)
- Known limitations & next steps
- How tests are run

---

8. Quick checklist before submission

- [ ] Provide database migrations or instructions to create the DB
- [ ] Provide `.env.example` for secrets
- [ ] Populate README with `curl`/Postman examples
- [ ] Add seed script to create sample logs
- [ ] Add screenshots of the dashboard (optional) or deploy screenshot

---

If you want, I can:

- Generate **actual project files** for backend and frontend (ready to `git clone` / run) and provide them in a ZIP.
- Produce a single `docker-compose` scaffold that runs everything locally.
- Provide a minimal working **zip** containing a small dataset and a working demo.

Tell me which of the three you'd like next and I will scaffold it for you (backend only / frontend only / full stack ZIP).