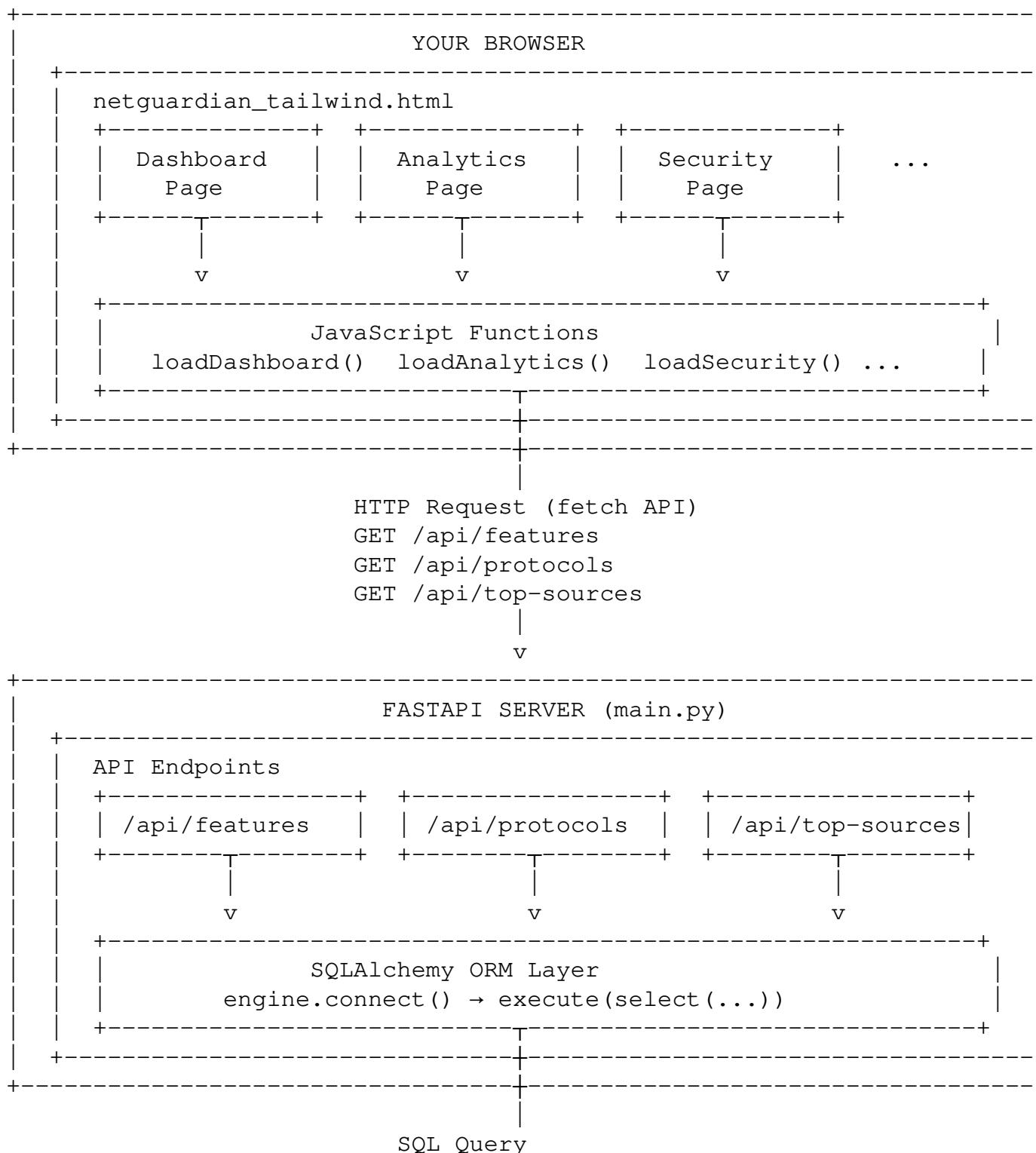


Dashboard Data Flow: Complete Beginner's Guide

A detailed explanation of how KPIs are queried from the database, the integration between frontend, backend, and database, and performance optimization strategies.

Architecture Overview



```

+
+-----+-----+-----+
|          DATABASE (SQLite/PostgreSQL)          |
+-----+-----+-----+
| Tables:                                     |
+-----+-----+-----+
| raw_packets      | aggregated_features | detected_alerts |
| (50+ columns)   | (40+ columns)    | (12 columns)   |
+-----+-----+-----+
+
+-----+

```

The Complete Data Flow

Phase 1: Initial Page Load

1. User opens browser → navigates to `http://localhost:5000/dashboard`
2. Browser sends GET request to FastAPI server
3. FastAPI's `dashboard_page()` function reads `netguardian_tailwind.html`
4. Server returns HTML file to browser
5. Browser parses HTML and loads JavaScript
6. JavaScript calls `loadDashboard()` automatically on page load

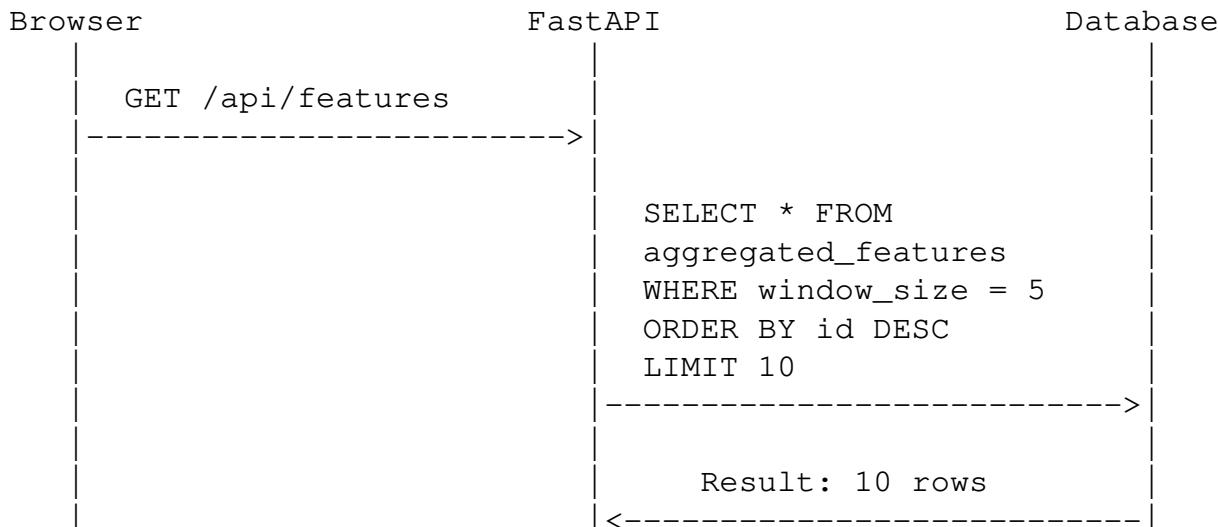
Phase 2: Data Fetching (The Main Loop)

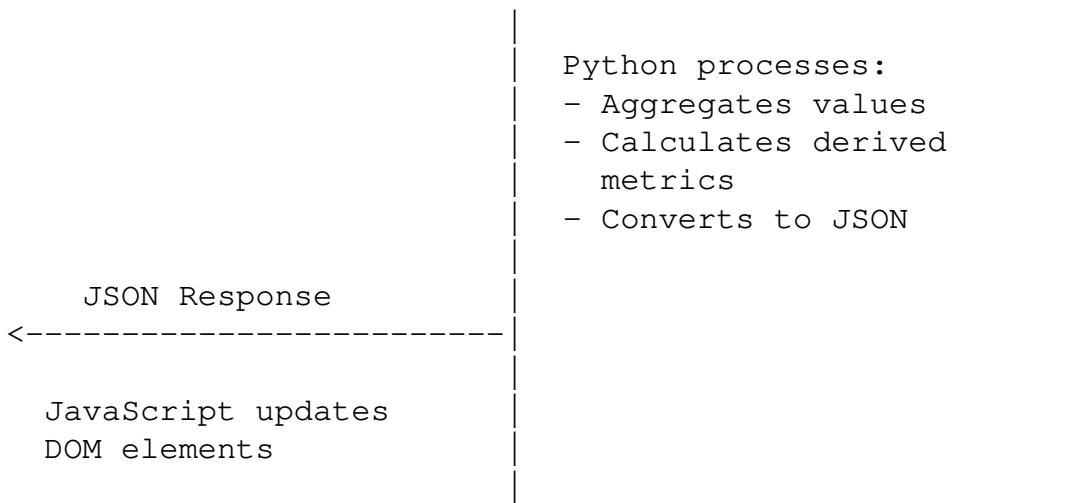
```

// This runs every 5 seconds automatically
setInterval(async () => {
    await loadDashboard();           // Fetches /api/features
    await loadProtocolChart();       // Fetches /api/protocols
    await loadTopSources();          // Fetches /api/top-sources
    await loadTopDestinations();     // Fetches /api/top-destinations
    await loadTopPorts();            // Fetches /api/top-ports
}, 5000);

```

Phase 3: Single API Request Lifecycle





Step-by-Step: What Happens When You Click

Example: Clicking the "Analytics" Tab

```

+-----+
| Step 1: USER CLICKS "Analytics" Tab
+-----+
|
| V
+-----+
| Step 2: JavaScript Event Handler Fires
|
| onclick="showPage('analytics')"
|
| function showPage(pageName) {
|   // Hide all pages
|   document.querySelectorAll('.page').forEach(page => {
|     page.classList.remove('active');
|   });
|   // Show selected page
|   document.getElementById('analytics-page').classList.add(...);
|   // Load data for that page
|   if (pageName === 'analytics') {
|     loadAnalytics(); // <-- This triggers the API call
|   }
| }
+-----+
|
| V
+-----+
| Step 3: loadAnalytics() Function Executes
|
| async function loadAnalytics() {
|   // Make HTTP request to backend
|   const response = await fetch('/api/features');
|   const features = await response.json();
| }
+-----+

```

This code snippet details the three steps involved in clicking the "Analytics" tab:

- Step 1: USER CLICKS "Analytics" Tab**
- Step 2: JavaScript Event Handler Fires**
- Step 3: loadAnalytics() Function Executes**

The code shows the implementation of the JavaScript event handler and the asynchronous function that triggers the API call.

```
// Calculate derived metrics in JavaScript
const health = calculateNetworkHealth(features);
const quality = calculateConnectionQuality(features);

// Update DOM with values
document.getElementById('analyticsHealth').textContent = ...;
```

}

|
v

Step 4: Browser Creates HTTP Request

```
GET /api/features HTTP/1.1
Host: localhost:5000
Accept: application/json
```

|
v

Step 5: FastAPI Receives Request & Routes to Handler

```
@app.get("/api/features")
def api_features(window_size: int = 5):
    conn = engine.connect()
    sel = select(aggregated_features_table).where(
        aggregated_features_table.c.window_size == window_size
    ).order_by(...).limit(10)
    result = conn.execute(sel).mappings().all()
    conn.close()
    return process_results(result)
```

|
v

Step 6: SQLAlchemy Generates SQL Query

```
SELECT id, src_ip, window_start, window_end, window_size,
       packet_count, packet_rate_pps, byte_count, byte_rate_bps,
       avg_packet_size, tcp_count, udp_count, icmp_count, ...
FROM aggregated_features
WHERE window_size = 5
ORDER BY id DESC
LIMIT 10
```

|
v

Step 7: Database Executes Query

1. Receives SQL query
2. Query planner determines best execution path
3. Scans aggregated_features table (or uses index)

```
4. Filters: WHERE window_size = 5
5. Sorts: ORDER BY id DESC
6. Limits: LIMIT 10
7. Returns 10 rows to SQLAlchemy
```

|
v

Step 8: Python Processes Results

```
# Aggregate across all 10 records
features = {
    'packet_count': sum(r['packet_count'] for r in records),
    'tcp_count': sum(r['tcp_count'] for r in records),
    'byte_rate_bps': sum(r['byte_rate_bps'] for r in records),
    ...
}
return features # FastAPI auto-converts to JSON
```

|
v

Step 9: HTTP Response Sent to Browser

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
    "packet_count": 15420,
    "byte_rate_bps": 1250000,
    "tcp_count": 12000,
    "udp_count": 2500,
    ...
}
```

|
v

Step 10: JavaScript Updates the UI

```
// Parse JSON response
const features = await response.json();

// Calculate derived metrics
const health = calculateNetworkHealth(features);

// Update DOM elements
document.getElementById('analyticsHealth').textContent = health;
document.getElementById('analyticsQuality').textContent = quality;

// User sees updated values on screen!
```

API Endpoints Explained

Current API Endpoints & Their Queries

Endpoint	Table Queried	Query Pattern	Rows Returned
/api/features	aggregated_features	LIMIT 10 with filter	10 rows
/api/protocols	raw_packets	LIMIT 1000 + Python grouping	1000 rows processed
/api/top-sources	raw_packets	LIMIT 1000 + Python grouping	1000 rows processed
/api/top-destinations	raw_packets	LIMIT 1000 + Python grouping	1000 rows processed
/api/top-ports	raw_packets	LIMIT 1000 + Python grouping	1000 rows processed
/api/alerts	detected_alerts	LIMIT 50	50 rows
/api/raw_packets/last/{N}	raw_packets	LIMIT N	N rows

Performance Analysis

Does This Overload the Database?

Short Answer: Not with SQLite for development, but YES with PostgreSQL at scale.

Current Load Analysis

Every 5 seconds, the dashboard makes these queries:

```
+-----+  
| Dashboard Page Refresh (Every 5 seconds) |  
+-----+  
| 1. GET /api/features      → 1 query, 10 rows |  
| 2. GET /api/protocols     → 1 query, 1000 rows + Python loop |  
| 3. GET /api/top-sources   → 1 query, 1000 rows + Python loop |  
| 4. GET /api/top-destinations → 1 query, 1000 rows + Python loop |  
| 5. GET /api/top-ports     → 1 query, 1000 rows + Python loop |  
+-----+  
| TOTAL: 5 queries per refresh |  
| ~4000 rows fetched from database |  
| All processing done in Python (not SQL) |  
+-----+
```

Problem Areas

Problem 1: Redundant Data Fetching

```
# These 4 endpoints ALL query the same 1000 rows:
```

```

/api/protocols      → SELECT * FROM raw_packets LIMIT 1000
/api/top-sources   → SELECT * FROM raw_packets LIMIT 1000 # SAME!
/api/top-destinations → SELECT * FROM raw_packets LIMIT 1000 # SAME!
/api/top-ports     → SELECT * FROM raw_packets LIMIT 1000 # SAME!

# That's 4000 rows fetched when 1000 would suffice!

```

➊ Problem 2: Processing Done in Python, Not SQL

```

# Current approach (SLOW):
packets = conn.execute(select(raw_packets).limit(1000)).all()
for p in packets:                      # Loop through 1000 rows in Python
    ip_counts[p['src_ip']] = ip_counts.get(p['src_ip'], 0) + 1
sorted_ips = sorted(ip_counts.items(), key=lambda x: x[1])

# Better approach (FAST):
# Let the database do the counting!
SELECT src_ip, COUNT(*) as count
FROM raw_packets
GROUP BY src_ip
ORDER BY count DESC
LIMIT 5

```

➋ Problem 3: No Caching

```

User A opens dashboard → 5 queries
User B opens dashboard → 5 queries (same data!)
User C opens dashboard → 5 queries (same data!)

```

With 10 users, you have 50 queries every 5 seconds = 600 queries/minute

➌ Problem 4: No Indexing

```

-- These queries are slow without indexes:
SELECT * FROM raw_packets ORDER BY id DESC LIMIT 1000 -- Full table scan
SELECT * FROM aggregated_features WHERE window_size = 5 -- No index!

```

What to Avoid

✗ DON'T: Fetch All Columns When You Need Few

```

# BAD - fetches 50+ columns when you only need 2
sel = select(raw_packets_table).limit(1000)

# GOOD - fetch only what you need
sel = select(raw_packets_table.c.src_ip, raw_packets_table.c.id).limit(1000)

```

✗ DON'T: Process Data in Python When SQL Can Do It

```

# BAD - fetch 1000 rows, loop in Python
packets = conn.execute(select(raw_packets).limit(1000)).all()

```

```

ip_counts = {}
for p in packets:
    ip_counts[p['src_ip']] = ip_counts.get(p['src_ip'], 0) + 1

# GOOD - let database do the work
from sqlalchemy import func
sel = select(
    raw_packets_table.c.src_ip,
    func.count().label('count')
).group_by(raw_packets_table.c.src_ip).order_by(func.count().desc()).li

```

X DON'T: Make Redundant API Calls

```

// BAD - same data fetched 4 times
await fetch('/api/features'); // for dashboard
await fetch('/api/features'); // for analytics
await fetch('/api/features'); // for security
await fetch('/api/features'); // for charts

// GOOD - fetch once, reuse
const features = await fetch('/api/features').then(r => r.json());
updateDashboard(features);
updateAnalytics(features);
updateSecurity(features);

```

X DON'T: Use Short Polling Intervals Without Need

```

// BAD - fetches every 1 second (720 queries/hour per user)
setInterval(loadDashboard, 1000);

// GOOD - 5 seconds is reasonable (144 queries/hour per user)
setInterval(loadDashboard, 5000);

// BETTER - use WebSockets for real-time, push only when data changes

```

X DON'T: Forget Connection Pooling

```

# BAD - creates new connection every request
conn = engine.connect() # Opens connection
# ... do work ...
conn.close() # Closes connection

# GOOD - use connection pooling (SQLAlchemy does this by default)
# Just make sure pool settings are configured:
engine = create_engine(
    DATABASE_URL,
    pool_size=10, # Keep 10 connections ready
    max_overflow=20, # Allow 20 more if needed
    pool_timeout=30, # Wait 30s for connection
    pool_recycle=1800 # Recycle connections every 30min
)

```

Best Practices & Improvements

✓ 1. Use SQL Aggregations

```
# Instead of fetching raw packets and counting in Python:  
from sqlalchemy import func  
  
@app.get("/api/top-sources")  
def api_top_sources(limit: int = 5):  
    conn = engine.connect()  
  
    # Let PostgreSQL do the counting!  
    query = select(  
        raw_packets_table.c.src_ip,  
        func.count().label('packet_count')  
    ).group_by(  
        raw_packets_table.c.src_ip  
    ).order_by(  
        func.count().desc()  
    ).limit(limit)  
  
    result = conn.execute(query).all()  
    conn.close()  
  
    return [{"ip": row.src_ip, "count": row.packet_count} for row in re
```

✓ 2. Add Database Indexes

```
-- Add these indexes for faster queries:  
CREATE INDEX idx_raw_packets_src_ip ON raw_packets(src_ip);  
CREATE INDEX idx_raw_packets_dst_ip ON raw_packets(dst_ip);  
CREATE INDEX idx_raw_packets_timestamp ON raw_packets(timestamp);  
CREATE INDEX idx_agg_features_window ON aggregated_features(window_size
```

✓ 3. Implement Caching

```
from functools import lru_cache  
import time  
  
# Simple time-based cache  
cache = {}  
CACHE_TTL = 5 # seconds  
  
@app.get("/api/features")  
def api_features():  
    cache_key = "features"  
    now = time.time()  
  
    # Return cached if fresh  
    if cache_key in cache and (now - cache[cache_key]['time']) < CACHE_  
        return cache[cache_key]['data']
```

```

# Fetch from database
data = fetch_features_from_db()

# Cache the result
cache[cache_key] = {'data': data, 'time': now}
return data

```

✓ 4. Combine Related Endpoints

```

# Instead of 5 separate endpoints, create one that returns everything:
@app.get("/api/dashboard-data")
def api_dashboard_data():
    return {
        "features": get_aggregated_features(),
        "protocols": get_protocol_distribution(),
        "top_sources": get_top_sources(),
        "top_destinations": get_top_destinations(),
        "top_ports": get_top_ports()
    }

// Frontend fetches once:
const data = await fetch('/api/dashboard-data').then(r => r.json());
updateFeatures(data.features);
updateProtocolChart(data.protocols);
updateTopSources(data.top_sources);
// ... etc

```

✓ 5. Use WebSockets for Real-Time Updates

```

from fastapi import WebSocket

@app.websocket("/ws/dashboard")
async def websocket_dashboard(websocket: WebSocket):
    await websocket.accept()
    while True:
        data = get_dashboard_data()
        await websocket.send_json(data)
        await asyncio.sleep(5) # Push every 5 seconds

// Frontend listens for updates:
const ws = new WebSocket('ws://localhost:5000/ws/dashboard');
ws.onmessage = (event) => {
    const data = JSON.parse(event.data);
    updateDashboard(data);
};

```

✓ 6. Paginate Large Datasets

```

@app.get("/api/raw_packets")
def api_raw_packets(page: int = 1, per_page: int = 50):
    offset = (page - 1) * per_page

    query = select(raw_packets_table).order_by(

```

```
    raw_packets_table.c.id.desc()
).offset(offset).limit(per_page)

return conn.execute(query).all()
```

Summary: Current vs Recommended Architecture

Aspect	Current	Recommended
Queries per refresh	5 separate	1 combined
Rows fetched	~4000	~100 (with SQL aggregation)
Processing	Python loops	SQL GROUP BY
Caching	None	5-second TTL cache
Indexing	None	Indexes on frequent columns
Real-time	Polling every 5s	WebSocket push
Connection management	Open/close per request	Connection pooling

Quick Reference: Performance Checklist

- [] Use SQL GROUP BY instead of Python loops
- [] Select only needed columns, not SELECT *
- [] Add indexes on filtered/sorted columns
- [] Implement server-side caching (5-10 second TTL)
- [] Combine multiple API calls into one
- [] Use connection pooling
- [] Consider WebSockets for real-time updates
- [] Paginate large result sets
- [] Monitor slow queries with EXPLAIN ANALYZE