```
# api_server.py
# Flask REST API server for network traffic analyzer

from flask import Flask, request, jsonify
from datetime import datetime, timedelta
import logging
from typing import Dict, List, Optional
import json

from database import DatabaseManager

class NetworkAnalyzerAPI:
    def __init__(self, db_config: Dict = None):
        """Initialize Flask API server"""
        self.app = Flask(__name__)
        self.db = DatabaseManager(**(db_config or {}))
        self.logger = logging.getLogger(__name__)

        # Setup routes
        self._setup_routes()

        # Configure logging
        logging.basicConfig(level=logging.INFO)

    def _setup_routes(self):
        """Setup API routes"""
```

```
@self.app.route('/api/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({
        'status': 'healthy',
        'timestamp': datetime.now().isoformat(),
        'service': 'network-traffic-analyzer'
    })

@self.app.route('/api/packets', methods=['GET'])
def get_packets():
    """Get packets with optional filtering"""
    try:
        # Parse query parameters
        filters = {}

        if request.args.get('source_ip'):
            filters['source_ip'] = request.args.get('source_ip')

        if request.args.get('destination_ip'):
            filters['destination_ip'] = request.args.get('destination_ip')

        if request.args.get('protocol'):
            filters['protocol'] = request.args.get('protocol').upper()
```

```
if request.args.get('start_time'):
    try:
        filters['start_time'] = datetime.fromisoformat(
            request.args.get('start_time').replace('Z', '+00:00')
        )
    except ValueError:
        return jsonify({'error': 'Invalid start_time format'}), 400

if request.args.get('end_time'):
    try:
        filters['end_time'] = datetime.fromisoformat(
            request.args.get('end_time').replace('Z', '+00:00')
        )
    except ValueError:
        return jsonify({'error': 'Invalid end_time format'}), 400

# Get limit parameter
limit = min(int(request.args.get('limit', 1000)), 10000)  # Max 10k records

# Retrieve packets from database
packets = self.db.get_packets(filters, limit)

# Convert datetime objects to ISO format for JSON serialization
for packet in packets:
    if packet.get('timestamp'):
        packet['timestamp'] = packet['timestamp'].isoformat()
```

```
            return jsonify({
                'packets': packets,
                'count': len(packets),
                'filters': filters,
                'limit': limit
            })

        except Exception as e:
            self.logger.error(f"Error retrieving packets: {e}")
            return jsonify({'error': 'Internal server error'}), 500

    @self.app.route('/api/alerts', methods=['GET'])
    def get_alerts():
        """Get alerts with optional filtering"""
        try:
            # Parse query parameters
            filters = {}

            if request.args.get('alert_type'):
                filters['alert_type'] = request.args.get('alert_type')

            if request.args.get('source_ip'):
                filters['source_ip'] = request.args.get('source_ip')

            if request.args.get('severity'):
```

```
    try:
        filters['severity'] = int(request.args.get('severity'))
    except ValueError:
        return jsonify({'error': 'Invalid severity value'}), 400

if request.args.get('resolved'):
    filters['resolved'] = request.args.get('resolved').lower() == 'true'

if request.args.get('start_time'):
    try:
        filters['start_time'] = datetime.fromisoformat(
            request.args.get('start_time').replace('Z', '+00:00')
        )
    except ValueError:
        return jsonify({'error': 'Invalid start_time format'}), 400

if request.args.get('end_time'):
    try:
        filters['end_time'] = datetime.fromisoformat(
            request.args.get('end_time').replace('Z', '+00:00')
        )
    except ValueError:
        return jsonify({'error': 'Invalid end_time format'}), 400

# Get limit parameter
limit = min(int(request.args.get('limit', 1000)), 10000)  # Max 10k records
```

```
        # Retrieve alerts from database
        alerts = self.db.get_alerts(filters, limit)

        # Convert datetime objects to ISO format for JSON serialization
        for alert in alerts:
            if alert.get('timestamp'):
                alert['timestamp'] = alert['timestamp'].isoformat()

        return jsonify({
            'alerts': alerts,
            'count': len(alerts),
            'filters': filters,
            'limit': limit
        })

    except Exception as e:
        self.logger.error(f"Error retrieving alerts: {e}")
        return jsonify({'error': 'Internal server error'}), 500

@self.app.route('/api/alerts/<int:alert_id>/resolve', methods=['POST'])
def resolve_alert(alert_id):
    """Mark an alert as resolved"""
    try:
        query = "UPDATE alerts SET resolved = true WHERE id = %s"
        with self.db.get_connection() as conn:
```

```
            with conn.cursor() as cursor:
                cursor.execute(query, (alert_id,))
                if cursor.rowcount == 0:
                    return jsonify({'error': 'Alert not found'}), 404
                conn.commit()

        return jsonify({
            'message': f'Alert {alert_id} marked as resolved',
            'alert_id': alert_id
        })

    except Exception as e:
        self.logger.error(f"Error resolving alert: {e}")
        return jsonify({'error': 'Internal server error'}), 500

@self.app.route('/api/stats', methods=['GET'])
def get_statistics():
    """Get traffic statistics"""
    try:
        # Get time range parameter (default 24 hours)
        time_range_hours = int(request.args.get('hours', 24))

        # Get traffic stats
        traffic_stats = self.db.get_traffic_stats(time_range_hours)

        # Get alert statistics
```

```
alert_query = """
SELECT alert_type, COUNT(*) as count, AVG(severity) as avg_severity
FROM alerts
WHERE timestamp >= NOW() - INTERVAL %s
GROUP BY alert_type
ORDER BY count DESC
"""

alert_stats = []
with self.db.get_connection() as conn:
    with conn.cursor() as cursor:
        cursor.execute(alert_query, (f"{time_range_hours} hours",))
        columns = [desc[0] for desc in cursor.description]
        alert_stats = [dict(zip(columns, row)) for row in cursor.fetchall()]

# Get top source IPs
top_ips_query = """
SELECT source_ip, COUNT(*) as packet_count
FROM packets
WHERE timestamp >= NOW() - INTERVAL %s
GROUP BY source_ip
ORDER BY packet_count DESC
LIMIT 10
"""

top_ips = []
```

```
        with self.db.get_connection() as conn:
            with conn.cursor() as cursor:
                cursor.execute(top_ips_query, (f"{time_range_hours} hours",))
                columns = [desc[0] for desc in cursor.description]
                top_ips = [dict(zip(columns, row)) for row in cursor.fetchall()]

        return jsonify({
            'time_range_hours': time_range_hours,
            'traffic_stats': traffic_stats,
            'alert_stats': alert_stats,
            'top_source_ips': top_ips,
            'generated_at': datetime.now().isoformat()
        })

    except Exception as e:
        self.logger.error(f"Error retrieving statistics: {e}")
        return jsonify({'error': 'Internal server error'}), 500

@self.app.route('/api/packets/search', methods=['POST'])
def search_packets():
    """Advanced packet search with complex filters"""
    try:
        data = request.get_json()
        if not data:
            return jsonify({'error': 'No search criteria provided'}), 400
```

```
            # Build dynamic query
            conditions = []
            params = {}

            if 'ip_range' in data:
                # Search for packets in IP range
                conditions.append(
                    "(source_ip::inet <<= %(ip_range)s OR destination_ip::inet <<= %(ip_range)s)"
                )
                params['ip_range'] = data['ip_range']

            if 'port_range' in data:
                port_min, port_max = data['port_range']
                conditions.append(
                    "((source_port BETWEEN %(port_min)s AND %(port_max)s) OR "
                    "(destination_port BETWEEN %(port_min)s AND %(port_max)s))"
                )
                params['port_min'] = port_min
                params['port_max'] = port_max

            if 'protocols' in data:
                conditions.append("protocol = ANY(%(protocols)s)")
                params['protocols'] = data['protocols']

            if 'tcp_flags' in data:
                conditions.append("tcp_flags LIKE %(tcp_flags)s")
```

```
    params['tcp_flags'] = f"%{data['tcp_flags']}%"

if 'packet_size_range' in data:
    size_min, size_max = data['packet_size_range']
    conditions.append("packet_size BETWEEN %(size_min)s AND %(size_max)s")
    params['size_min'] = size_min
    params['size_max'] = size_max

if not conditions:
    return jsonify({'error': 'No valid search criteria provided'}), 400

# Build and execute query
base_query = """
SELECT id, timestamp, source_ip, destination_ip, source_port, destination_port,
       protocol, packet_size, tcp_flags, node_id
FROM packets
WHERE """ + " AND ".join(conditions) + """
ORDER BY timestamp DESC
LIMIT %(limit)s
"""

params['limit'] = min(data.get('limit', 1000), 10000)

with self.db.get_connection() as conn:
    with conn.cursor() as cursor:
        cursor.execute(base_query, params)
```

```
                columns = [desc[0] for desc in cursor.description]
                packets = [dict(zip(columns, row)) for row in cursor.fetchall()]

            # Convert datetime objects to ISO format
            for packet in packets:
                if packet.get('timestamp'):
                    packet['timestamp'] = packet['timestamp'].isoformat()

            return jsonify({
                'packets': packets,
                'count': len(packets),
                'search_criteria': data
            })

        except Exception as e:
            self.logger.error(f"Error in packet search: {e}")
            return jsonify({'error': 'Internal server error'}), 500

    @self.app.route('/api/nodes', methods=['GET'])
    def get_nodes():
        """Get information about active capture nodes"""
        try:
            query = """
            SELECT
                node_id,
                COUNT(*) as packet_count,
```

```
    MIN(timestamp) as first_seen,
    MAX(timestamp) as last_seen,
    COUNT(DISTINCT protocol) as protocols_seen
FROM packets
WHERE timestamp >= NOW() - INTERVAL '24 hours'
GROUP BY node_id
ORDER BY packet_count DESC
"""

with self.db.get_connection() as conn:
    with conn.cursor() as cursor:
        cursor.execute(query)
        columns = [desc[0] for desc in cursor.description]
        nodes = [dict(zip(columns, row)) for row in cursor.fetchall()]

# Convert datetime objects to ISO format
for node in nodes:
    if node.get('first_seen'):
        node['first_seen'] = node['first_seen'].isoformat()
    if node.get('last_seen'):
        node['last_seen'] = node['last_seen'].isoformat()

return jsonify({
    'nodes': nodes,
    'count': len(nodes)
})
```

```
            except Exception as e:
                self.logger.error(f"Error retrieving node information: {e}")
                return jsonify({'error': 'Internal server error'}), 500

        @self.app.errorhandler(404)
        def not_found(error):
            return jsonify({'error': 'Endpoint not found'}), 404

        @self.app.errorhandler(500)
        def internal_error(error):
            return jsonify({'error': 'Internal server error'}), 500

    def run(self, host='0.0.0.0', port=5000, debug=False):
        """Run the Flask application"""
        self.logger.info(f"Starting API server on {host}:{port}")
        self.app.run(host=host, port=port, debug=debug)

def main():
    """Main function for standalone execution"""
    import argparse

    # Setup logging
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
```

```
    )

    # Parse command line arguments
    parser = argparse.ArgumentParser(description='Network Traffic Analyzer - REST API Server')
    parser.add_argument('--host', default='0.0.0.0', help='Host to bind the server')
    parser.add_argument('--port', type=int, default=5000, help='Port to bind the server')
    parser.add_argument('--debug', action='store_true', help='Enable debug mode')
    parser.add_argument('--db-host', default='localhost', help='Database host')
    parser.add_argument('--db-port', type=int, default=5432, help='Database port')
    parser.add_argument('--db-name', default='network_analyzer', help='Database name')
    parser.add_argument('--db-user', default='postgres', help='Database user')
    parser.add_argument('--db-password', default='password', help='Database password')

    args = parser.parse_args()

    # Database configuration
    db_config = {
        'host': args.db_host,
        'port': args.db_port,
        'database': args.db_name,
        'user': args.db_user,
        'password': args.db_password
    }

    try:
        # Initialize and start API server
```

```
        api = NetworkAnalyzerAPI(db_config)
        api.run(host=args.host, port=args.port, debug=args.debug)

    except Exception as e:
        print(f"Error starting API server: {e}")

if __name__ == "__main__":
    main()
```