IP LOOKUP TOOL - DETAILED DOCUMENTATION

TABLE OF CONTENTS

1. PROJECT OVERVIEW

The IP Lookup Tool is a comprehensive solution for IP address analysis, providing both geolocation and threat intelligence data. It's designed with scalability and extensibility in mind, using modern Python practices and asynchronous programming.

Key Features:

- Multi-provider support for redundancy and comprehensive data
- Asynchronous operations for improved performance
- Extensible architecture for easy addition of new providers
- Configurable API timeouts and error handling
- Data export capabilities

2. ARCHITECTURE

Directory Structure:

```
ip_lookup_tool/
    api/                    - API integration layer
    core/                   - Core business logic
    reporting/              - Data export functionality
    ui/                     - User interface
    config.py              - Configuration management
    main.py                - Application entry point
    requirements.txt       - Dependencies
```

Design Patterns:

- Provider Pattern: Abstract base class for API providers
- Factory Pattern: Provider instantiation
- Strategy Pattern: Provider selection
- Singleton Pattern: Configuration management

3. CORE COMPONENTS

A. API Layer (api/)

- `base_provider.py`
  - Abstract base class defining provider interface
  - Common functionality for all providers
  - Error handling and response validation
- `geo_providers.py`
  - Implementation of geolocation providers
  - Supports ipinfo.io and ip-api.com
  - Each provider implements BaseProvider interface
  - Handles provider-specific data transformation
- `threat_providers.py`
  - Implementation of threat intelligence providers
  - Similar structure to geo_providers
  - Focuses on security-related data

B. Core Layer (core/)

- `orchestrator.py`
  - Main orchestration logic
  - Manages provider selection and data aggregation
  - Handles concurrent API requests
  - Implements error handling and retry logic
- `ip_utils.py`
  - IP address validation
  - Format conversion
  - CIDR notation handling
  - IP range calculations

C. Reporting Layer (reporting/)

- `exporter.py`
  - Data export functionality
  - Supports multiple formats (JSON, CSV)
  - Customizable output templates
  - Batch processing capabilities

D. UI Layer (ui/)

- `app_gui.py`
  - Graphical user interface
  - Real-time data display
  - Interactive provider selection
  - Export options

4. API PROVIDERS

A. Geolocation Providers

IPInfo Provider:

- Endpoint: ipinfo.io
- Features:
  - City, region, country data
  - Latitude/longitude coordinates
  - Organization information
  - Hostname resolution
- Rate limits: Based on API key tier
- Data format: JSON

IP-API Provider:

- Endpoint: ip-api.com
- Features:
  - Free tier available
  - Basic geolocation data
  - ISP information
  - Reverse DNS lookup
- Rate limits: 45 requests per minute (free tier)
- Data format: JSON

5. CONFIGURATION

`config.py` contains:

API Configuration:

```
IPINFO_API_KEY = "your_api_key"
API_TIMEOUT = 10  # seconds
```

Provider Settings:

```
DEFAULT_PROVIDERS = ["ipinfo", "ipapi.com"]
MAX_RETRIES = 3
```

Export Settings:

```
DEFAULT_EXPORT_FORMAT = "json"
EXPORT_DIRECTORY = "exports"
```

6. USAGE GUIDE

Basic Usage:

```
from core.orchestrator import IPLookupOrchestrator
```

```
# Initialize
orchestrator = IPLookupOrchestrator()

# Single IP lookup
results = await orchestrator.lookup_ip("8.8.8.8")

# Batch lookup
ip_list = ["8.8.8.8", "1.1.1.1"]
batch_results = await orchestrator.lookup_multiple_ips(ip_list)
```

Advanced Usage:

```
# Custom provider selection
results = await orchestrator.lookup_ip(
    "8.8.8.8",
    providers=["ipinfo"]
)

# Export results
from reporting.exporter import Exporter
exporter = Exporter()
exporter.export(results, format="csv")
```

7. TECHNICAL DETAILS

A. Dependencies

aiohttp:

- Purpose: Asynchronous HTTP client/server
- Version: >=3.8.0
- Key features:
  - Async/await support
  - Connection pooling
  - Timeout handling
  - SSL/TLS support

Additional Dependencies:

- `aiodns`: Asynchronous DNS resolution
- `typing`: Type hints support
- Other dependencies in requirements.txt

B. Error Handling

- Provider-specific error handling
- Retry mechanism for failed requests
- Fallback providers
- Detailed error logging

C. Performance Considerations

- Connection pooling
- Caching mechanisms
- Batch processing
- Rate limit handling

D. Security

- API key management
- SSL/TLS for all requests
- Input validation
- Rate limiting

E. Best Practices

1. Always use async/await for API calls
2. Implement proper error handling
3. Use connection pooling
4. Monitor rate limits
5. Cache frequently requested data
6. Validate input data
7. Use type hints for better code maintainability

CONTRIBUTING

Development Setup:

1. Clone the repository
2. Create a virtual environment
3. Install dependencies
4. Set up API keys
5. Run tests

Code Style:

- Follow PEP 8 guidelines
- Use type hints
- Document all public methods
- Write unit tests

Testing:

- Unit tests for each component
- Integration tests for providers
- Performance testing
- Error handling tests