Bugchella Audit Tool - Developer Guide

This guide explains how to extend the Bugchella Audit Tool with new functionality. It's intended for developers who want to add features or modify existing capabilities.

Project Structure

```
bugchella audit tool/
├── cli.py                   # Main CLI entry point
├── api_auth.py              # API authentication
├── data_sources.py          # Data fetching functions
├── requirements.txt         # Dependencies
├── .env                     # API credentials (not in repo)
├── audits/                  # Audit modules
│   ├── properties.py        # Property-related audits
│   ├── customers.py         # Customer-related audits
│   ├── assets.py            # Asset-related audits
│   └── vendors.py           # Vendor-related audits
└── README.md                # Project documentation
```

Architecture Overview

The tool follows a modular architecture:

1. **CLI Layer** (`cli.py`): Handles command-line arguments and calls appropriate audit functions
2. **API Authentication** (`api_auth.py`): Manages authentication with the BuildOps API
3. **Data Sources** (`data_sources.py`): Provides functions to fetch data from the API
4. **Audit Modules** (`audits/`): Contains domain-specific audit logic organized by entity type

Adding a New Audit Function

To add a new audit function, follow these steps:

1. Choose the Appropriate Module

Determine which entity your audit relates to (properties, customers, assets, or vendors) and locate the corresponding file in the `audits/` directory.

2. Implement the Audit Function

Add your function to the appropriate module. Use existing functions as templates.

Example audit function pattern:

```python
def audit_my_new_feature():
    """
    Docstring explaining what the audit does.
    Returns a dict with result data.
    """
    # Fetch necessary data
    data = fetch_required_data()

    # Process the data
    results = []
    for item in data:
        if meets_your_criteria(item):
            results.append((item['name'], item['id']))

    # Return results in a consistent format
    return {
        "items": results,
        "totalCount": len(results)
    }
```

3. Add CLI Command

Open `cli.py` and add a new command to expose your audit function:

```
@app.command()
def my_new_command():
    """Help text describing what your command does."""
    result = audit_my_new_feature()

    # Format and display results
    print("Results of my new audit:")
    print("----------------------")
    for name, id in result['items']:
        print(f"{name} (ID: {id})")

    print(f"\nTotal: {result['totalCount']} items found")
```

Working with Data Sources

The `data_sources.py` file contains functions for fetching data from the API. If you need to access a new type of data:

1. Add a New Data Fetching Function

```
def get_my_new_data(limit=100, page=0):
    """
    Fetch my new data type from the API.
    """
    url = f"{API_BASE_URL}/my-new-endpoint"
    params = {
        "limit": limit,
        "page_size": page_size,
    }
    response = make_authenticated_request(url, params=params)
    return response.json()
```

2. Create a Cached Fetcher Function (Optional)

For data that will be accessed frequently, consider creating a cached fetch function in the appropriate audit module:

```python
@cache
def fetch_all_my_data(limit=100):
    """
    Fetch all my data items.
    Returns a list of items.
    """
    max_workers = 50
    items = []
    first_page = get_my_new_data(limit=limit, page=0)
    total_count = first_page.get('totalCount', 0)
    items = first_page.get('items', [])

    if total_count <= limit:
        return items

    num_pages = (total_count + limit - 1) // limit
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = [
            executor.submit(get_my_new_data, limit=limit, page=page)
            for page in range(1, num_pages)
        ]
        for future in as_completed(futures):
            items.extend(future.result().get('items', []))

    return items
```

Performance Considerations

Parallel Processing

For operations that involve multiple API requests or heavy processing:

1. Use `ThreadPoolExecutor` for I/O-bound operations (like API calls)
2. Use `ProcessPoolExecutor` for CPU-bound operations

Caching

1. Use the `@cache` decorator for functions that return the same result when called with the same parameters
2. Consider implementing more sophisticated caching for frequently used data

Testing Your Extensions

1. **Manual Testing**: Run your new command with different inputs
2. **Automated Testing**: Consider adding tests to verify your audit logic

Documentation

When adding new features:

1. Update the `README.md` with your new commands
2. Add a section to the `User_Guide.md` explaining how to use your feature
3. Include detailed docstrings in your code

Common Issues and Solutions

- **API Rate Limiting**: If you encounter rate limiting, implement exponential backoff in your requests
- **Memory Issues**: For large datasets, consider streaming or pagination approaches
- **Performance**: Use profiling to identify bottlenecks and optimize accordingly

Getting Help

If you need assistance with the BuildOps API:

- Refer to the API documentation
- Contact the BuildOps API support team

For issues with the Bugchella Audit Tool codebase:

- Examine existing audit modules for patterns and examples
- Reach out to the project maintainers