
YAML Filesystem Trees

Jifeng Wu
jifengwu2k@gmail.com

Abstract

We introduce YAML filesystem trees, a compact, human- and machine-readable representation of file and directory trees in YAML. This notation aims to offer intuitive editing and precise semantics, facilitating diverse applications: environment specification, package manifests, deployment scripts, reproducible research, and more. We describe the format, enumerate its core properties, and provide a reference implementation for recursive traversal; we also discuss applications such as copying, validation, extraction, and visualization.

1 Introduction

Many workflows require the explicit, platform-independent description of directory and file hierarchies. Existing tools - tar, rsync, package managers - operate on filesystem snapshots but lack a simple, editable, and verifiable intermediate form. While JSON and YAML are often used for configuration, there is no widely adopted, sufficiently expressive serialization for arbitrary filesystem trees that is both human- and machine-friendly.

We present a *YAML filesystem tree* format that addresses this gap. The notation directly models the containment hierarchy of files and directories, is trivial to parse (PyYAML), and is friendly to version control and code review. The structure is not tied to any particular action (copying, deployment), but may be used wherever unambiguous, composable file tree descriptions are needed.

2 YAML Filesystem Trees

Structure and Semantics

- **Files** are represented as JSON-quoted strings - simply their names.
- **Directories** are single-key dictionaries, with the key the JSON-quoted directory name.
- The directory's value is a list of contents (files and subdirectories), or omitted (i.e., `null`) for an empty directory.
- All entries live in order within a list.

Example

```
- "empty-dir":  
- "file-1"  
- "src":  
  - "main.py"  
  - "lib":  
    - "util.py"  
    - "README.md"
```

This specifies a directory containing an empty directory `empty-dir`, a file `file-1`, and a subtree rooted at `src/`.

Key Properties

- **Human-editable:** Intuitive indented tree, close to `ls -R` or Unix `tree` output.
- **Script-friendly:** Loads as a Python structure (list of strings/dicts) without further transformation.
- **Declarative:** Describes only topology, enabling many possible realizations.

3 Applications

This notation is broadly valuable for:

- **Reproducible pipelines:** Define the complete state of a test or deploy environment: Version-control setup and expected result trees alongside code and data.
- **Archiving and packaging:** Use as a manifest for building archives, installers, etc.
- **Bulk operations:** Serve as the basis for copy/move/extract operations (see below).

4 Traversal

Traversing YAML filesystem trees recursively is trivial. Here is a reference Python function, parameterized for actions such as copying, validation, or tree walking. For concreteness, we illustrate *copying files* from a source to a target, but the same logic can be applied to other operations.

```
import os, shutil
from typing import List, Union, Dict, Optional, Tuple

def walk_yaml_filesystem_tree(
    yaml_filesystem_tree: List[Union[str, Dict[str, Optional[List]]]],
    action_fn,
    relpath: Tuple[str, ...] = ()
) -> None:
    """
    Recursively walk the tree, calling action_fn(name, is_dir, relpath)
    """
    for item in yaml_filesystem_tree:
        if isinstance(item, str):
            action_fn(item, is_dir=False, relpath=relpath)
        elif isinstance(item, dict) and len(item) == 1:
            dirname = next(iter(item))
            children = item[dirname]
            action_fn(dirname, is_dir=True, relpath=relpath)
            if isinstance(children, list): # Non-empty dir
                walk_yaml_filesystem_tree(
                    children,
                    action_fn,
                    relpath + (dirname,)
                )
        else:
            raise ValueError(f"Malformed entry: {item}")
```

A copy realization (see Appendix) can use this generic walker.

5 Discussion

The YAML filesystem tree offers a sweet spot between rigidity and readability. By constraining the structure but not the usage, it is suitable as an "API surface" between tools, between maintainers and users, or between automated steps in computational pipelines. Deterministic parsing and intuitive editing foster trust and minimize "configuration drift" errors.

6 Conclusion

We have introduced, motivated, and formalized YAML filesystem trees, a minimalist notation for arbitrary directory/file trees. This notation is appropriate for a broad spectrum of filesystem-aware workflows, from build automation to reproducible research. A demonstration traversal framework is given, suitable for direct application or further extension.

Appendix: Example - Copying Files and Making Directories

One concrete use case is copying a tree from SOURCE_DIR to TARGET_DIR. This is realized by providing an `action_fn` as follows:

```
import os
import shutil

source_dir = ...          # (Set externally)
target_dir = ...

def copy_action(name, is_dir, relpath_components):
    path_src = os.path.join(source_dir, *relpath_components, name)
    path_dst = os.path.join(target_dir, *relpath_components, name)
    if is_dir:
        os.makedirs(path_dst, exist_ok=True)
    else:
        shutil.copy2(path_src, path_dst)
```

`walk_yaml_tree(yaml_tree, copy_action)` then realizes copying, but alternative actions (validation, printing, etc.) are easily implemented.