Project File Contents

Generated: 2025-07-22 01:05:29

Existing branches: main
Current branch: `main`

Commit: `2caf255`

Repo: https://github.com/sdkng44/document_folder_structure

Status: □□ synced

Index

---

docs\document_folder_structure.html

(Total lines: 2023) (Showing up to 10 lines, max 2000 characters)

```
1 | <!doctype html>
2 | <html lang="en">
3 | <head>
4 |     <meta charset="utf-8">
5 |     <meta name="viewport" content="width=device-width, initial-scale=1">
6 |     <meta name="generator" content="pdoc 15.0.4"/>
7 |     <title>document_folder_structure API documentation</title>
8 |
9 |     <style>/*! * Bootstrap Reboot v5.0.0 (https://getbootstrap.com/) * Copyright 2011-202
```

---

docs\index.html

(Total lines: 7) (Showing up to 10 lines, max 2000 characters)

```
1 | <!doctype html>
2 | <html>
3 | <head>
4 |     <meta charset="utf-8">
5 |     <meta http-equiv="refresh" content="0; url=./document_folder_structure.html"/>
6 | </head>
7 | </html>
```

---

docs\search.js

(Total lines: 46) (Showing up to 10 lines, max 2000 characters)

```
1 | window.pdocSearch = (function(){
2 | /** elasticlunr - http://weixsong.github.io * Copyright (C) 2017 Oliver Nightingale * Cop
```

LICENSE

(Total lines: 201) (Showing up to 10 lines, max 2000 characters)

```
1 |                                 Apache License
2 |                           Version 2.0, January 2004
3 |                        http://www.apache.org/licenses/
4 |
5 |    TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
6 |
7 |    1. Definitions.
8 |
9 |       "License" shall mean the terms and conditions for use, reproduction,
10 |       and distribution as defined by Sections 1 through 9 of this document.
```

README.md

(Total lines: 126) (Showing up to 10 lines, max 2000 characters)

```
1 | # document_folder_structure
2 | ## **Automatic Markdown Documentation and Directory Tree Generator**
3 |
4 | This script generates structured Markdown documentation for any project directory, includ
5 |
6 | - **directory_tree.md:** Directory structure with Git metadata (branch, commit, repo, etc
7 | - **all_files_content.md:** Grouped file index and content previews (with truncation for
8 | - Skips configured folders/files (see configuration)
9 | - Automatically adds the documentation folder to `.gitignore`
10 | - Supports GitHub/GitLab links to each file at current commit (if Git is configured)
```

---

config.json

(Total lines: 14)

```
1 | {
2 |     "excluded_dirs": ["node_modules", "__pycache__", ".git", "deps", ".fingerprint", "bui
3 |     "excluded_files": [".gitignore", ".849C9593-D756-4E56-8D6E-42412F2A707B"],
4 |     "excluded_extensions": [".log", ".tmp", ".bak", ".db-shm", ".db-wal", ".npmrc", ".pre
5 |     "max_depth": 4,
6 |     "truncate_files": ["README", "LICENSE", "CHANGELOG", "CONTRIBUTING", "SECURITY"],
7 |     "truncate_exts": [".md", ".txt"],
8 |     "truncate_dirs": ["docs"],
9 |     "truncate_file_pairs": [["document_folder_structure", ".html"]],
10 |     "max_truncate_chars": 2000,
11 |     "truncate_lines": 10,
12 |     "max_log_lines": 10,
```

```
13 | "max_preview_columns": 20
14 | }
```

---

document_folder_structure.py

(Total lines: 748)

```
1 | """
2 | Markdown Documentation and Directory Tree Generator for Projects.
3 |
4 | Usage example:
5 |     python document_folder_structure.py C:\\path\\to\\your\\project
6 |
7 | Creates two files in ./INTERNAL_DOCS/:
8 |     - directory_tree.md (directory structure with Git info)
9 |     - all_files_content.md (index and file contents, binaries omitted)
10 |
11 | Includes Git metadata (branch, commit, branches, repo) and GitHub links (if applicable).
12 | Skips configurable folders/files (README, LICENSE, CHANGELOG, CONTRIBUTING, SECURITY).
13 | Automatically adds created folder to .gitignore.
14 |
15 | Sample config file, first 4 lines are for exclude files and directories before the tree
16 | {
17 |     "excluded_dirs": ["node_modules", "__pycache__", ".git", "deps", ".fingerprint", "bu
18 |     "excluded_files": [".gitignore", ".849C9593-D756-4E56-8D6E-42412F2A707B"],
19 |     "excluded_extensions": [".log", ".tmp", ".bak", ".db-shm", ".db-wal", ".npmrc", ".pr
20 |     "max_depth": 4,
21 |     "truncate_files": ["README", "LICENSE", "CHANGELOG", "CONTRIBUTING", "SECURITY"],
22 |     "truncate_exts": [".md", ".txt"],
23 |     "truncate_dirs": ["docs"],
24 |     "truncate_file_pairs": [["document_folder_structure", ".html"]],
25 |     "max_truncate_chars": 2000,
26 |     "truncate_lines": 10,
27 |     "max_log_lines": 10,
28 |     "max_preview_columns": 20
29 | }
30 |
31 | Requirements:
32 |     - Python 3.6+
33 |     - Run "pip install -r requirements.txt" to install dependencies.
34 |
35 | """
36 | import csv
37 | try:
38 |     import openpyxl
39 | except ImportError:
40 |     openpyxl = None
41 | import os
42 | import json
43 | import logging
44 | import datetime
45 | import subprocess
46 | from argparse import ArgumentParser
47 |
48 | DEFAULTS = {
49 |     "excluded_dirs": ["node_modules", "__pycache__", ".git", "deps", ".fingerprint", "bu
50 |     "excluded_files": [".gitignore", ".849C9593-D756-4E56-8D6E-42412F2A707B"],
51 |     "excluded_extensions": [".log", ".tmp", ".bak", ".db-shm", ".db-wal", ".npmrc", ".pr
52 |     "max_depth": 4,
53 |     "truncate_files": ["README", "LICENSE", "CHANGELOG", "CONTRIBUTING", "SECURITY"],
54 |     "truncate_exts": [".md", ".txt"],
55 |     "truncate_dirs": ["docs"],
```

```
    "truncate_file_pairs": [["document_folder_structure", ".html"]],
    "max_truncate_chars": 2000,
    "truncate_lines": 10,
    "max_log_lines": 10,
    "max_preview_columns": 20
}

# === SETUP LOGGING ===
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)

def add_line_numbers(lines, start=1):
    """
    Adds line numbers to each line in a list.
    """
    return [f"{i + start} | {line.rstrip()}\n" for i, line in enumerate(lines)]


def load_config_with_defaults(config_file, args):
    """
    Load config file.
    """
    config = DEFAULTS.copy()
    # Load config file
    if os.path.exists(config_file):
        with open(config_file, 'r') as f:
            file_config = json.load(f)
        config.update(file_config)
    # Load args
    if getattr(args, "max_depth", None) is not None:
        config["max_depth"] = args.max_depth
    if getattr(args, "truncate_lines", None) is not None:
        config["truncate_lines"] = args.truncate_lines
    if getattr(args, "truncate_chars", None) is not None:
        config["max_truncate_chars"] = args.truncate_chars
    if getattr(args, "max_log_lines", None) is not None:
        config["max_log_lines"] = args.max_log_lines
    if getattr(args, "max_preview_columns", None) is not None:
        config["max_preview_columns"] = args.max_preview_columns
    return config


def read_n_lines_max_chars(filepath, config, max_lines=None, max_chars=None):
    """
    Reads up to `max_lines` lines and never more than `max_chars` characters.
     Returns a list of lines (not joined), possibly truncated.
     """

     if max_lines is None:
         max_lines = config["truncate_lines"]
     if max_chars is None:
         max_chars = config["max_truncate_chars"]

     lines = []
     char_count = 0
     try:
         with open(filepath, 'r', encoding='utf-8', errors='replace') as fin:
             for i, line in enumerate(fin):
                 if i >= max_lines or char_count >= max_chars:
                     break
                 # Only add up to remaining allowed characters for this line
                 remaining = max_chars - char_count
                 if len(line) > remaining:
                     lines.append(line[:remaining] + " ... [TRUNCATED]\n")
                     char_count += remaining
                     break
                 else:
                     lines.append(line)
```

```
                    char_count += len(line)
    except Exception as e:
        lines = [f"(Could not read file: {e})\n"]
    return lines


def preview_csv(filepath, config, max_lines=None):
    """
    Returns the first lines of a CSV as a Markdown table.
    """

    if max_lines is None:
        max_lines = config["max_log_lines"]
    max_columns = config["max_preview_columns"]
    lines = []
    try:
        with open(filepath, newline='', encoding='utf-8', errors='replace') as csvfile:
            reader = csv.reader(csvfile)
            headers = next(reader, None)
            if headers:
                if len(headers) > max_columns:
                    lines.append(f"_(Only showing first {max_columns} columns)_")
                headers = headers[:max_columns]
                lines.append("| " + " | ".join(headers) + " |")
                lines.append("|" + "|".join("---" for _ in headers) + "|")
            count = 0
            for row in reader:
                row = row[:max_columns]
                lines.append("| " + " | ".join(row) + " |")
                count += 1
                if count >= max_lines:
                    break
    except Exception as e:
        lines = [f"(Could not read CSV: {e})"]
    return lines

def preview_excel(filepath, config, max_lines=None):
    """
    Returns the first lines of each sheet in an Excel (.xlsx) file as a Markdown table.
    """
    max_columns = config["max_preview_columns"]
    if max_lines is None:
        max_lines = config["max_log_lines"]


    if openpyxl is None:
        return ["(openpyxl is not installed, cannot preview Excel)"]
    lines = []
    try:
        wb = openpyxl.load_workbook(filepath, read_only=True)
        for sheet in wb.sheetnames:
            ws = wb[sheet]
            rows = list(ws.iter_rows(values_only=True))
            if not rows:
                continue
            lines.append(f"### Sheet: {sheet}")
            headers = [str(h) if h is not None else "" for h in rows[0][:max_columns]]
            if len(rows[0]) > max_columns:
                lines.append(f"_(Only showing first {max_columns} columns)_")
            lines.append("| " + " | ".join(headers) + " |")
            lines.append("|" + "|".join("---" for _ in headers) + "|")
            for i, row in enumerate(rows[1:max_lines+1]):
                cells = [str(cell) if cell is not None else "" for cell in row[:max_col
                lines.append("| " + " | ".join(cells) + " |")
            lines.append("")
    except Exception as e:
```

```
        lines = [f"(Could not read Excel: {e})"]
    return lines


def ensure_gitignore_has_internal_docs(project_dir):
    """
    Ensures that the INTERNAL_DOCS/ folder is included in the .gitignore file.

    If .gitignore does not exist, it is created. If it exists but the line is missing,

    Args:
        project_dir (str): Path to the project root directory.
    """
    gitignore_path = os.path.join(project_dir, '.gitignore')
    try:
        if not os.path.exists(gitignore_path):
            with open(gitignore_path, 'w', encoding='utf-8') as f:
                f.write('INTERNAL_DOCS/\n')
            return
        with open(gitignore_path, 'r', encoding='utf-8') as f:
            lines = f.read().splitlines()
        if 'INTERNAL_DOCS/' not in lines:
            with open(gitignore_path, 'a', encoding='utf-8') as f:
                with open(gitignore_path, 'rb') as fr:
                    fr.seek(-1, os.SEEK_END)
                    last_char = fr.read(1)
                if last_char != b'\n':
                    f.write('\n')
                f.write('INTERNAL_DOCS/\n')
    except Exception as e:
        print(f"Warning: could not update .gitignore: {e}")


def ensure_internal_docs_dir(base_path):
    """
    Creates the INTERNAL_DOCS folder if it does not exist.

    Args:
        base_path (str): Base path to create the folder in.

    Returns:
        str: Absolute path to the INTERNAL_DOCS folder.
    """
    internal_docs = os.path.join(base_path, "INTERNAL_DOCS")
    if not os.path.exists(internal_docs):
        os.makedirs(internal_docs)
    return internal_docs

def count_tree_stats(directory, config, max_depth=None, depth=0):
    """
    Counts the total files and folders in the tree, respecting exclusions and depth.

    Args:
        directory (str): Path to the root directory.
        config (dict): Configuration (exclusions).
        max_depth (int, optional): Maximum depth to traverse.
        depth (int, optional): Current depth.

    Returns:
        (int, int): Tuple (files, folders)
    """
    total_files, total_dirs = 0, 0
    try:
        files = os.listdir(directory)
    except Exception:
        return 0, 0
```

```
    for file in files:
        path = os.path.join(directory, file)
        if os.path.isdir(path):
            if file not in config['excluded_dirs']:
                total_dirs += 1
                if max_depth is None or depth < max_depth:
                    f, d = count_tree_stats(path, config, max_depth, depth + 1)
                    total_files += f
                    total_dirs += d
        else:
            if file in config['excluded_files'] or file.endswith(tuple(config['excluded
                continue
            total_files += 1
    return total_files, total_dirs

def is_binary(filename, blocksize=512):
    """
    Heuristically detects if a file is binary.

    Args:
        filename (str): File path.

    Returns:
        bool: True if the file appears to be binary.
    """
    try:
        with open(filename, 'rb') as f:
            chunk = f.read(blocksize)
            if b'\0' in chunk:
                return True
            try:
                chunk.decode('utf-8')
                return False
            except UnicodeDecodeError:
                return True
    except Exception:
        return True
    return False

def get_github_url(git_info, filepath):
    """
    Builds a GitHub (or GitLab) URL for a given file and commit.

    Args:
        git_info (tuple): (branch, commit, status, remote_url)
        filepath (str): Relative file path.

    Returns:
        str: GitHub URL (or empty if not applicable).
    """
    if not git_info or not git_info[3] or not git_info[1]:
        return ""
    remote = git_info[3]
    commit = git_info[1]
    # Supports SSH and HTTPS formats (GitHub and GitLab)
    if remote.startswith("git@"):
        remote = remote.replace(":", "/").replace("git@", "https://").replace(".git", "
    elif remote.startswith("https://"):
        remote = remote.replace(".git", "")
    return f"{remote}/blob/{commit}/{filepath.replace(os.sep, '/')}"

def get_git_info(directory):
    """
    Gets Git information for the repository in the given directory.

    Args:
        directory (str): Path to the repo root directory.
```

```
    Returns:
        tuple: (branch, commit, status, remote_url), or (None, None, None, None) if not
    """
    try:
        branch = subprocess.check_output(
            ["git", "rev-parse", "--abbrev-ref", "HEAD"],
            cwd=directory, stderr=subprocess.DEVNULL
        ).decode().strip()
        commit = subprocess.check_output(
            ["git", "rev-parse", "--short", "HEAD"],
            cwd=directory, stderr=subprocess.DEVNULL
        ).decode().strip()
        status = subprocess.check_output(
            ["git", "status", "--porcelain"],
            cwd=directory, stderr=subprocess.DEVNULL
        ).decode()
        dirty = "□□ synced" if not status else "□□ local changes (need to push to branc
        remote = subprocess.check_output(
            ["git", "config", "--get", "remote.origin.url"],
            cwd=directory, stderr=subprocess.DEVNULL
        ).decode().strip()
        return branch, commit, dirty, remote
    except Exception:
        return None, None, None, None

def get_git_branches(directory):
    """
    Returns a list of existing branches in the repo.

    Args:
        directory (str): Repo path.

    Returns:
        list: List of branch names, empty if not a repo.
    """
    try:
        out = subprocess.check_output(
            ["git", "branch", "--format", "%(refname:short)"], cwd=directory
        ).decode().strip().split("\n")
        out = [x.strip() for x in out if x.strip()]
        return out
    except Exception:
        return []

# Extension dictionary for code highlighting in Markdown code blocks
LANG_EXT = {
    ".py": "python", ".md": "markdown", ".json": "json", ".js": "javascript",
    ".ts": "typescript", ".tsx": "tsx", ".jsx": "jsx", ".html": "html", ".css": "css",
    ".scss": "scss", ".sass": "sass", ".less": "less", ".yml": "yaml", ".yaml": "yaml",
    ".toml": "toml", ".sh": "bash", ".bat": "bat", ".ps1": "powershell", ".txt": "",
    ".xml": "xml", ".csv": "csv", ".ini": "ini", ".conf": "conf", ".php": "php", ".rb":
    ".go": "go", ".java": "java", ".c": "c", ".cpp": "cpp", ".h": "cpp", ".hpp": "cpp",
    ".env": "", ".sql": "sql", ".log": ""
}
"""
    Dictionary of possible file extensions for Markdown code highlighting.
"""

# Always exclude generated documentation files
DOC_OUTPUTS = {"directory_tree.md", "all_files_content.md"}

def generate_tree_and_collect_files(
    directory, prefix="", depth=0, max_depth=None, is_last=True, config=None,
    files_list=None, extensions_set=None, exclude_outputs=True, base_docs_path=None
):
    """
```

```
    Generates the directory tree (as text) and collects the list of valid files.

    Args:
        directory (str): Project root.
        prefix (str): Visual prefix for branches.
        depth (int): Current depth.
        max_depth (int): Maximum depth to traverse.
        is_last (bool): Whether this is the last item at this level.
        config (dict): Exclusion configuration.
        files_list (list): List to add found files to.
        extensions_set (set): Set of detected extensions.
        exclude_outputs (bool): Whether to exclude generated docs.
        base_docs_path (str): Base path for relative paths.

    Returns:
        list: Lines of the tree.
    """
    if max_depth is not None and depth > max_depth:
        return []
    try:
        files = os.listdir(directory)
    except PermissionError as e:
        logging.warning(f"Permission denied: {e}")
        return []
    files.sort()
    tree = []
    excluded_dirs = config['excluded_dirs']
    excluded_files = set(config['excluded_files'])
    excluded_extensions = tuple(config['excluded_extensions'])
    for index, file in enumerate(files):
        path = os.path.join(directory, file)
        is_last_item = index == len(files) - 1
        connector = "└── " if is_last_item else "├── "
        if os.path.isdir(path):
            if file in excluded_dirs:
                tree.append(f"{prefix}{connector}+ {file}/ (excluded)")
            else:
                tree.append(f"{prefix}{connector}+ {file}/")
                tree.extend(
                    generate_tree_and_collect_files(
                        path, prefix + ("    " if is_last_item else "│   "),
                        depth + 1, max_depth, is_last_item, config, files_list, extensi
                    )
                )
        else:
            rel_file = os.path.relpath(path, base_docs_path)
            if (
                file in excluded_files or
                file.endswith(excluded_extensions) or
                (exclude_outputs and file in DOC_OUTPUTS and os.path.dirname(path).ends
            ):
                continue
            tree.append(f"{prefix}{connector}- {file}")
            if files_list is not None:
                files_list.append(os.path.relpath(path, base_docs_path))
            if extensions_set is not None:
                ext = os.path.splitext(file)[1].lower()
                extensions_set.add(ext)
    return tree

def markdown_tree_header(title, total_files, total_dirs, git_info=None, extensions=None
    """
    Builds the header for the directory tree, with git metadata and extensions.

    Args:
        title (str): Title.
        total_files (int): Number of files.
```

```
        total_dirs (int): Number of folders.
        git_info (tuple): Git info.
        extensions (set): Detected extensions.
        branches (list): Repo branches.

    Returns:
        str: Markdown header.
    """
    now = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    header = f"# {title}\n\n_Generated: {now}_\n\n"
    if branches:
        header += f"**Existing branches:** {', '.join(branches)}  \n"
    if git_info and git_info[0]:
        header += (
            f"**Current branch:** `{git_info[0]}`  \n"
            f"**Commit:** `{git_info[1]}`  \n"
            f"**Repo:** {git_info[3]}  \n"
            f"**Status:** {git_info[2]}\n\n"
        )
    if extensions:
        exts = ", ".join(sorted(ext for ext in extensions if ext))
        header += f"**Detected extensions:** `{exts}`\n\n" if exts else ""
    header += f"_(Files: {total_files}, Folders: {total_dirs})_\n\n```text\n"
    return header

def markdown_tree_footer():
    """Returns the footer for the tree block (code block closing)."""
    return "```\n"

def print_tree_and_collect_files(
    directory, config, output_file="directory_tree.md", tree_title="Project Directory T
):
    """
    Prints the tree in Markdown and returns the list of found files and git_info.

    Returns:
        files_list (list): Found files.
        git_info (tuple): Git info of the repo.
    """
    max_depth = config.get('max_depth')
    files_list = []
    extensions_set = set()
    tree = generate_tree_and_collect_files(
        directory, max_depth=max_depth, config=config,
        files_list=files_list, extensions_set=extensions_set,
        exclude_outputs=True, base_docs_path=directory
    )
    total_files, total_dirs = count_tree_stats(directory, config, max_depth)
    git_info = get_git_info(directory)
    git_branches = get_git_branches(directory)
    header = markdown_tree_header(tree_title, total_files, total_dirs, git_info, extens
    footer = markdown_tree_footer()
    with open(output_file, 'w', encoding='utf-8') as f:
        f.write(header)
        for line in tree:
            f.write(line + "\n")
        f.write(footer)
    return files_list, git_info

def group_files_by_folder(files_list):
    """
    Converts a list of files into a hierarchical tree (nested dicts) by folder.
    """
    tree = {}
    for path in files_list:
        parts = path.split(os.sep)
        cur = tree
```

```
        for i, part in enumerate(parts):
            if i == len(parts) - 1:
                cur.setdefault("_files", []).append(path)
            else:
                cur = cur.setdefault(part, {})
    return tree

def render_index(tree, basepath="", git_info=None, level=0):
    """
    Renders the grouped index in Markdown from the folder/file tree.

    Args:
        tree (dict): Tree generated by group_files_by_folder.
        basepath (str): Current base path.
        git_info (tuple): Git info for links.
        level (int): Indentation.

    Returns:
        list: Markdown lines.
    """
    lines = []
    indent = "    " * level
    folders = [k for k in tree if k != "_files"]
    for folder in folders:
        lines.append(f"{indent}- **{folder}/**")
        lines.extend(render_index(tree[folder], os.path.join(basepath, folder), git_inf
    if "_files" in tree:
        for filepath in tree["_files"]:
            anchor = filepath.replace("/", "").replace(".", "").replace("\\", "")
            url = get_github_url(git_info, filepath)
            fname = os.path.basename(filepath)
            link_line = f"{indent}    - [{fname}](#{anchor})"
            if url:
                link_line += f" | [GitHub]({url})"
            lines.append(link_line)
    return lines




def should_truncate_file(filepath, config):
    """
    Determines if a file should be truncated in preview, by name/extension/path.
    """

    for d in config["truncate_dirs"]:
        parts = [p.lower() for p in filepath.replace("\\", "/").split("/")]
        if d.lower() in parts:
            return True

    base = os.path.splitext(os.path.basename(filepath))[0]
    ext = os.path.splitext(filepath)[1].lower()
    for pair in config["truncate_file_pairs"]:
        if base == pair[0] and ext == pair[1]:
            return True

    if base.upper() in (t.upper() for t in config["truncate_files"]):
        return True

    if ext in config["truncate_exts"]:
        return True

    return False


def flatten_files_in_tree(tree):
```

```
    """
    Returns an ordered list of all files in the hierarchical tree.
    """
    files = []
    folders = [k for k in tree if k != "_files"]
    for folder in folders:
        files.extend(flatten_files_in_tree(tree[folder]))
    if "_files" in tree:
        files.extend(tree["_files"])
    return files

def generate_content_document(
    files_list, output_file, git_info, config, base_docs_path=None, project_dir=None, n
):
    """
    Generates the content document with index and preview/truncation of each file.

    Args:
        files_list (list): List of files.
        output_file (str): Output file.
        git_info (tuple): Git info.
        config: Config.json file.
        base_docs_path (str): Root for paths.
        project_dir (str): For git branch info.

    Effect: writes the Markdown content file.
    """
    now = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    out_lines = [f"# Project File Contents\n\n_Generated: {now}_\n\n"]
    branches = get_git_branches(project_dir)
    if branches:
        out_lines.append(f"**Existing branches:** {', '.join(branches)}  \n")
    if git_info and git_info[0]:
        out_lines.append(
            f"**Current branch:** `{git_info[0]}`  \n"
            f"**Commit:** `{git_info[1]}`   \n"
            f"**Repo:** {git_info[3]}  \n"
            f"**Status:** {git_info[2]} \n"
        )
    out_lines.append("## Index\n")
    index_tree = group_files_by_folder(files_list)
    out_lines.extend(render_index(index_tree, "", git_info, level=0))
    out_lines.append("\n---\n")
    ordered_files = flatten_files_in_tree(index_tree)
    for file_path in ordered_files:
        ext = os.path.splitext(file_path)[1].lower()
        language = LANG_EXT.get(ext, "")
        anchor = file_path.replace("/", "").replace(".", "").replace("\\", "")
        out_lines.append(f"## {file_path}\n<a name=\"{anchor}\"></a>\n")
        abs_path = file_path if not base_docs_path else os.path.join(base_docs_path, fi
        if os.path.exists(abs_path) and not is_binary(abs_path):
            try:
                with open(abs_path, 'r', encoding='utf-8', errors='replace') as f:
                    total_lines = sum(1 for _ in f)
                out_lines.append(f"_(Total lines: {total_lines})_\n")
            except Exception:
                pass
        if is_binary(abs_path):
            out_lines.append("_(Binary file omitted)_\n")
            out_lines.append(f"```{language}\n```\n")
        elif ext == ".csv":
            out_lines.append("_(CSV preview)_\n")
            out_lines.append(f"_(Showing first {config['max_log_lines']} rows)_\n")
            out_lines.extend(preview_csv(abs_path, config))
            out_lines.append("\n")
        elif ext == ".xlsx":
            out_lines.append("_(Excel preview)_\n")
```

```
            out_lines.append(f"_(Showing first {config['max_log_lines']} rows)_\n")
            out_lines.extend(preview_excel(abs_path, config))
            out_lines.append("\n")
        elif ext == ".xls":
            out_lines.append("_(Preview not supported for .xls files. Use .xlsx)_\n")
        elif ext == ".log":
            try:
                with open(abs_path, 'r', encoding='utf-8', errors='replace') as fin:
                    lines = fin.readlines()
                if len(lines) > config["max_log_lines"]:
                    out_lines.append(f"_(Showing last {config['max_log_lines']} lines)_
                numbered_lines = add_line_numbers(lines[-config["max_log_lines"]:], sta
                out_lines.append(f"```{language}\n")
                out_lines.extend(numbered_lines)
                out_lines.append("```\n")
            except Exception as e:
                logging.warning(f"Could not read {abs_path}: {e}")
                out_lines.append(f"(Could not read file: {e})\n")
        elif should_truncate_file(abs_path, config):
            try:
                lines = read_n_lines_max_chars(abs_path, config)
                if not no_lines:
                    lines = add_line_numbers(lines)
                else:
                    lines = [l if l.endswith('\n') else l + '\n' for l in lines]
                out_lines.append(f"_(Showing up to {config['truncate_lines']} lines, ma
                out_lines.append(f"```{language}\n")
                out_lines.extend(lines)
                out_lines.append("```\n")
            except Exception as e:
                logging.warning(f"Could not read {abs_path}: {e}")
                out_lines.append(f"(Could not read file: {e})\n")
        else:
            try:
                with open(abs_path, 'r', encoding='utf-8', errors='replace') as fin:
                    lines = fin.readlines()
                if not no_lines:
                    lines = add_line_numbers(lines)
                else:
                    lines = [l if l.endswith('\n') else l + '\n' for l in lines]
                out_lines.append(f"```{language}\n")
                out_lines.extend(lines)
                out_lines.append("```\n")
            except Exception as e:
                logging.warning(f"Could not read {abs_path}: {e}")
                out_lines.append(f"(Could not read file: {e})\n")
        if not out_lines[-1].endswith('\n'):
            out_lines.append('\n')
        out_lines.append("---\n")
    with open(output_file, 'w', encoding='utf-8') as out:
        out.write("".join(out_lines))


# ========== MAIN ==========
if __name__ == "__main__":
    parser = ArgumentParser(description="Generates directory tree and file content docu
    parser.add_argument("directory", help="Project root directory.")
    parser.add_argument("--config", default="config.json", help="JSON configuration fil
    parser.add_argument("--max-depth", type=int, help="Maximum tree depth.")
    parser.add_argument("--tree-title", default="Project Directory Tree", help="Title f
    parser.add_argument("--truncate-lines", type=int, help="Max lines for files like RE
    parser.add_argument("--truncate-chars", type=int, help="Max characters to show for
    parser.add_argument("--max-log-lines", type=int, help="Max lines to show for .log f
    parser.add_argument("--max-preview-columns", type=int, help="Max columns to show in
    parser.add_argument("--no-lines", action="store_true", help="Disable per-line numbe
    args = parser.parse_args()

```

```
726 |     config = load_config_with_defaults(args.config, args)
727 |
728 |     ensure_gitignore_has_internal_docs(args.directory)
729 |     output_dir = ensure_internal_docs_dir(args.directory)
730 |     tree_output_path = os.path.join(output_dir, "directory_tree.md")
731 |     content_output_path = os.path.join(output_dir, "all_files_content.md")
732 |     files_list, git_info = print_tree_and_collect_files(
733 |         args.directory,
734 |         config,
735 |         output_file=tree_output_path,
736 |         tree_title=args.tree_title
737 |     )
738 |     generate_content_document(
739 |         files_list,
740 |         output_file=content_output_path,
741 |         git_info=git_info,
742 |         config=config,
743 |         base_docs_path=args.directory,
744 |         project_dir=args.directory,
745 |         no_lines=args.no_lines
746 |     )
747 |     print(f"DONE -> Directory tree saved in {tree_output_path}")
748 |     print(f"DONE -> File contents saved in {content_output_path}")
```

---

requirements.txt

(Total lines: 1) (Showing up to 10 lines, max 2000 characters)

```
1 | openpyxl>=3.0.0
```
