File: requirements.txt

```
fpdf2
GitPython
inquirer
pathspec
pytest
```

File: pyproject.toml

```
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "repo2pdf"
version = "0.1.0"
description = "Convert coding repositories into PDFs and JSON summaries"
authors = [
  { name="Haris Sujethan", email="your-email@example.com" },
]
license = {text = "MIT"}
readme = "README.md"
requires-python = ">=3.7"
dependencies = [
  "fpdf2",
  "GitPython",
  "inquirer",
  "pathspec",
]

[project.scripts]
repo2pdf = "repo2pdf.cli:main"
```

File: README.md

repo-2-pdf

Convert your repositories into clean PDFs and structured JSON outputs, designed for AI ingestion pipelines but also useful for documentation.

Features

- Convert **local** or **remote GitHub repositories**
- Generate **PDFs** containing full file structures and contents
- Output structured **JSON summaries** for AI context ingestion
- Exclude unnecessary file types automatically

Installation

Option 1: Install from PyPI (Recommended)

```bash
pip install repo2pdf
```

Option 2: Install from Source

Clone the repository and install locally:

```bash
git clone https://github.com/haris-sujethan/repo-2-pdf
cd repo-2-pdf
pip install -r requirements.txt
```

Then choose one of the following:

Local development install (recommended):

```bash
pip install -e .
repo2pdf
```

Run without installing:

```bash
python -m repo2pdf.cli
```

Usage

Run the CLI tool:

```bash
repo2pdf
```

Follow the interactive prompts:

1. Select local or remote repository
2. Provide the local repo path or GitHub URL
3. Choose an output location
4. Exclude any file types you don't want included (e.g., `.png`, `.jpg`)
5. Optionally generate a JSON summary alongside the PDF

Example CLI Flow

The CLI provides an interactive terminal interface that guides you through the conversion process:

Example Outputs

Example outputs are available in the `/examples` folder:

- **repo_output.pdf** - Clean PDF with full repository structure and file contents
- **repo_output.json** - Structured JSON summary perfect for AI ingestion

File: setup.py

```
from setuptools import setup, find_packages

setup(
    name='repo2pdf',
    version='0.1.0',
    packages=find_packages(),
    install_requires=[
        'fpdf2',
        'GitPython',
        'inquirer'
    ],
    entry_points={
        'console_scripts': [
            'repo2pdf=repo2pdf.cli:main',
        ],
    },
)
```

File: .gitignore

```
repo2pdf.egg-info/
__pycache__/
*.py[cod]
*$py.class
```

File: .pytest_cache/CACHEDIR.TAG

Signature: 8a477f597d28d172789f06886806bc55
This file is a cache directory tag created by pytest.
For information about cache directory tags, see:
#https://bford.info/cachedir/spec.html

File: .pytest_cache/README.md

pytest cache directory

This directory contains data from the pytest's cache plugin,
which provides the `--lf` and `--ff` options, as well as the `cache` fixture.

Do not commit this to version control.

See [the docs](https://docs.pytest.org/en/stable/how-to/cache.html) for more information.

File: .pytest_cache/.gitignore

```
# Created by pytest automatically.
*
```

File: .pytest_cache/v/cache/nodeids

```
[
  "tests/test_core.py::test_process_local_repo_creates_outputs",
  "tests/test_core.py::test_process_remote_repo_clones_and_generates",
  "tests/test_core.py::test_traverse_repo_excludes_specified_files",
  "tests/test_core.py::test_traverse_repo_reads_files",
  "tests/test_pdf.py::test_generate_pdf_creates_file",
  "tests/test_utils.py::test_output_json_creates_valid_file"
]
```

File: .pytest_cache/v/cache/lastfailed

{}

File: .pytest_cache/v/cache/stepwise

[]

File: tests/test_utils.py

```
import os
import tempfile
import json
from repo2pdf.utils import output_json

def test_output_json_creates_valid_file():
    with tempfile.TemporaryDirectory() as tmpdir:
        output_path = os.path.join(tmpdir, "output.pdf")
        files = [("test.py", "print('hello')")]

        output_json(files, output_path)

        json_path = output_path.replace(".pdf", ".json")
        assert os.path.exists(json_path)

        with open(json_path) as f:
            data = json.load(f)
            assert "files" in data
            assert data["files"][0]["path"] == "test.py"
            assert "print('hello')" in data["files"][0]["content"]
```

File: tests/test_pdf.py

```
import os
import tempfile
from repo2pdf.pdf import generate_pdf

def test_generate_pdf_creates_file():
    with tempfile.TemporaryDirectory() as tmpdir:
        output_path = os.path.join(tmpdir, "output.pdf")
        files = [("test.py", "print('hello')")]

        generate_pdf(files, output_path)

        assert os.path.exists(output_path)
        assert os.path.getsize(output_path) > 0
```

File: tests/test_core.py

```
import os
import tempfile
from repo2pdf.core import traverse_repo
import os
import tempfile
from repo2pdf.core import process_local_repo

def test_traverse_repo_reads_files():
    with tempfile.TemporaryDirectory() as tmpdir:
        # Create a dummy file
        file_path = os.path.join(tmpdir, "test.py")
        with open(file_path, "w") as f:
            f.write("print('hello')")

        files = traverse_repo(tmpdir)

        assert len(files) == 1
        assert files[0][0] == "test.py"
        assert "print('hello')" in files[0][1]

def test_traverse_repo_excludes_specified_files():
    with tempfile.TemporaryDirectory() as tmpdir:
        # Create two files: one .py and one .png
        py_path = os.path.join(tmpdir, "test.py")
        png_path = os.path.join(tmpdir, "image.png")

        with open(py_path, "w") as f:
            f.write("print('hello')")

        with open(png_path, "w") as f:
            f.write("binarydata")

        from repo2pdf.core import traverse_repo
        files = traverse_repo(tmpdir)

        # Default traverse_repo (no exclude param) should return both files
        assert any(f[0] == "test.py" for f in files)

        # Now test excluding .png
        files_exclude = traverse_repo(tmpdir, exclude_list=[".png"])
        assert any(f[0] == "test.py" for f in files_exclude)
        assert not any(f[0] == "image.png" for f in files_exclude)

def test_process_remote_repo_clones_and_generates(monkeypatch):
    from repo2pdf.core import process_remote_repo
    import tempfile
    import os

    # Use a very small public GitHub repo for testing
    test_repo_url = "https://github.com/octocat/Hello-World.git"

    with tempfile.TemporaryDirectory() as tmpdir:
```

```
        output_path = os.path.join(tmpdir, "output.pdf")

        # Monkeypatch os.getcwd to tmpdir so output is saved there
        monkeypatch.setattr(os, "getcwd", lambda: tmpdir)

        # Run process_remote_repo with delete=True to clean up after test
            process_remote_repo(test_repo_url, want_json=True, output_path=output_path, exclude_list=[],
delete=True)

        assert os.path.exists(output_path)
        assert os.path.getsize(output_path) > 0

        json_path = output_path.replace(".pdf", ".json")
        assert os.path.exists(json_path)

def test_process_local_repo_creates_outputs(monkeypatch):
    with tempfile.TemporaryDirectory() as tmpdir:
        # Create a dummy local repo file
        file_path = os.path.join(tmpdir, "test.py")
        with open(file_path, "w") as f:
            f.write("print('hello')")

        output_path = os.path.join(tmpdir, "repo_output.pdf")

        # Monkeypatch os.getcwd to tmpdir so outputs are saved there
        monkeypatch.setattr(os, "getcwd", lambda: tmpdir)

        # Run process_local_repo with JSON generation
        process_local_repo(tmpdir, want_json=True)

        assert os.path.exists(output_path)
        assert os.path.getsize(output_path) > 0

        json_path = output_path.replace(".pdf", ".json")
        assert os.path.exists(json_path)
```

File: tests/__init__.py

File: .git/config

```
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
ignorecase = true
precomposeunicode = true
[remote "origin"]
url = https://github.com/haris-sujethan/repo-2-pdf.git
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
remote = origin
merge = refs/heads/main
```

File: .git/HEAD

ref: refs/heads/main

File: .git/description

Unnamed repository; edit this file 'description' to name the repository.

File: .git/COMMIT_EDITMSG

updating ASCII art

File: .git/info/exclude

```
# git ls-files --others --exclude-from=.git/info/exclude
# Lines that start with '#' are comments.
# For a project mostly in C, the following would be a good set of
# exclude patterns (uncomment them if you want to use them):
# *.[oa]
# *~
```

File: .git/logs/HEAD

0000000000000000000000000000000000000000 dd01af437f46da2a0732d0dc4e041a7c018fa0ef Haris Sujethan <sujethanharis@gmail.com> 1751761527 -0700commit (initial): repo2pdf MVP
dd01af437f46da2a0732d0dc4e041a7c018fa0ef 0000000000000000000000000000000000000000 Haris Sujethan <sujethanharis@gmail.com> 1751761537 -0700Branch: renamed refs/heads/master to refs/heads/main
0000000000000000000000000000000000000000 dd01af437f46da2a0732d0dc4e041a7c018fa0ef Haris Sujethan <sujethanharis@gmail.com> 1751761537 -0700Branch: renamed refs/heads/master to refs/heads/main
dd01af437f46da2a0732d0dc4e041a7c018fa0ef 99b0807b0b0f804ca4b066b4137ecdaaaf5b993a Haris Sujethan <sujethanharis@gmail.com> 1751762693 -0700commit: added ASCII art to the CLI
99b0807b0b0f804ca4b066b4137ecdaaaf5b993a b74b09fa2d7f4a2c275e690bebabbd6bf791751b Haris Sujethan <sujethanharis@gmail.com> 1751764489 -0700commit: added tests
b74b09fa2d7f4a2c275e690bebabbd6bf791751b 8529346910dd4f5a240b33f575eef56f1a8b750b Haris Sujethan <sujethanharis@gmail.com> 1751771220 -0700commit: updating ASCII art
8529346910dd4f5a240b33f575eef56f1a8b750b cdd15676887f405925ed6289b0515de8c8c96f6d Haris Sujethan <sujethanharis@gmail.com> 1751771360 -0700commit: updating ASCII art

File: .git/logs/refs/heads/main

0000000000000000000000000000000000000000 dd01af437f46da2a0732d0dc4e041a7c018fa0ef Haris Sujethan <sujethanharis@gmail.com> 1751761527 -0700commit (initial): repo2pdf MVP
dd01af437f46da2a0732d0dc4e041a7c018fa0ef dd01af437f46da2a0732d0dc4e041a7c018fa0ef Haris Sujethan <sujethanharis@gmail.com> 1751761537 -0700Branch: renamed refs/heads/master to refs/heads/main
dd01af437f46da2a0732d0dc4e041a7c018fa0ef 99b0807b0b0f804ca4b066b4137ecdaaaf5b993a Haris Sujethan <sujethanharis@gmail.com> 1751762693 -0700commit: added ASCII art to the CLI
99b0807b0b0f804ca4b066b4137ecdaaaf5b993a b74b09fa2d7f4a2c275e690bebabbd6bf791751b Haris Sujethan <sujethanharis@gmail.com> 1751764489 -0700commit: added tests
b74b09fa2d7f4a2c275e690bebabbd6bf791751b 8529346910dd4f5a240b33f575eef56f1a8b750b Haris Sujethan <sujethanharis@gmail.com> 1751771220 -0700commit: updating ASCII art
8529346910dd4f5a240b33f575eef56f1a8b750b cdd15676887f405925ed6289b0515de8c8c96f6d Haris Sujethan <sujethanharis@gmail.com> 1751771360 -0700commit: updating ASCII art

File: .git/logs/refs/remotes/origin/main

0000000000000000000000000000000000000000 dd01af437f46da2a0732d0dc4e041a7c018fa0ef Haris Sujethan <sujethanharis@gmail.com> 1751761544 -0700update by push
dd01af437f46da2a0732d0dc4e041a7c018fa0ef 99b0807b0b0f804ca4b066b4137ecdaaaf5b993a Haris Sujethan <sujethanharis@gmail.com> 1751762697 -0700update by push
99b0807b0b0f804ca4b066b4137ecdaaaf5b993a b74b09fa2d7f4a2c275e690bebabbd6bf791751b Haris Sujethan <sujethanharis@gmail.com> 1751764492 -0700update by push
b74b09fa2d7f4a2c275e690bebabbd6bf791751b 8529346910dd4f5a240b33f575eef56f1a8b750b Haris Sujethan <sujethanharis@gmail.com> 1751771223 -0700update by push
8529346910dd4f5a240b33f575eef56f1a8b750b cdd15676887f405925ed6289b0515de8c8c96f6d Haris Sujethan <sujethanharis@gmail.com> 1751771364 -0700update by push

File: .git/hooks/commit-msg.sample

```
#!/bin/sh
#
# An example hook script to check the commit log message.
# Called by "git commit" with one argument, the name of the file
# that has the commit message.  The hook should exit with non-zero
# status after issuing an appropriate message if it wants to stop the
# commit.  The hook is allowed to edit the commit message file.
#
# To enable this hook, rename this file to "commit-msg".

# Uncomment the below to add a Signed-off-by line to the message.
# Doing this in a hook is a bad idea in general, but the prepare-commit-msg
# hook is more suited to it.
#
# SOB=$(git var GIT_AUTHOR_IDENT | sed -n 's/^\(.*>\).*$/Signed-off-by: \1/p')
# grep -qs "^$SOB" "$1" || echo "$SOB" >> "$1"

# This example catches duplicate Signed-off-by lines.

test "" = "$(grep '^Signed-off-by: ' "$1" |
 sort | uniq -c | sed -e '/^[ ]*1[ ]/d')" || {
echo >&2 Duplicate Signed-off-by lines.
exit 1
}
```

File: .git/hooks/pre-rebase.sample

```
#!/bin/sh
#
# Copyright (c) 2006, 2008 Junio C Hamano
#
# The "pre-rebase" hook is run just before "git rebase" starts doing
# its job, and can prevent the command from running by exiting with
# non-zero status.
#
# The hook is called with the following parameters:
#
# $1 -- the upstream the series was forked from.
# $2 -- the branch being rebased (or empty when rebasing the current branch).
#
# This sample shows how to prevent topic branches that are already
# merged to 'next' branch from getting rebased, because allowing it
# would result in rebasing already published history.

publish=next
basebranch="$1"
if test "$#" = 2
then
topic="refs/heads/$2"
else
topic=`git symbolic-ref HEAD` ||
exit 0 ;# we do not interrupt rebasing detached HEAD
fi

case "$topic" in
refs/heads/??/*)
;;
*)
exit 0 ;# we do not interrupt others.
;;
esac

# Now we are dealing with a topic branch being rebased
# on top of master.  Is it OK to rebase it?

# Does the topic really exist?
git show-ref -q "$topic" || {
echo >&2 "No such branch $topic"
exit 1
}

# Is topic fully merged to master?
not_in_master=`git rev-list --pretty=oneline ^master "$topic"`
if test -z "$not_in_master"
then
echo >&2 "$topic is fully merged to master; better remove it."
exit 1 ;# we could allow it, but there is no point.
fi
```

```
# Is topic ever merged to next?  If so you should not be rebasing it.
only_next_1=`git rev-list ^master "^$topic" ${publish} | sort`
only_next_2=`git rev-list ^master           ${publish} | sort`
if test "$only_next_1" = "$only_next_2"
then
not_in_topic=`git rev-list "^$topic" master`
if test -z "$not_in_topic"
then
echo >&2 "$topic is already up to date with master"
exit 1 ;# we could allow it, but there is no point.
else
exit 0
fi
else
not_in_next=`git rev-list --pretty=oneline ^${publish} "$topic"`
/usr/bin/perl -e '
my $topic = $ARGV[0];
my $msg = "* $topic has commits already merged to public branch:\n";
my (%not_in_next) = map {
/^([0-9a-f]+) /;
($1 => 1);
} split(/\n/, $ARGV[1]);
for my $elem (map {
/^([0-9a-f]+) (.*)$/;
[$1 => $2];
} split(/\n/, $ARGV[2])) {
if (!exists $not_in_next{$elem->[0]}) {
if ($msg) {
print STDERR $msg;
undef $msg;
}
print STDERR " $elem->[1]\n";
}
}
' "$topic" "$not_in_next" "$not_in_master"
exit 1
fi

<<\DOC_END

This sample hook safeguards topic branches that have been
published from being rewound.

The workflow assumed here is:

 * Once a topic branch forks from "master", "master" is never
   merged into it again (either directly or indirectly).

 * Once a topic branch is fully cooked and merged into "master",
   it is deleted.  If you need to build on top of it to correct
   earlier mistakes, a new topic branch is created by forking at
   the tip of the "master".  This is not strictly necessary, but
   it makes it easier to keep your history simple.
```

 * Whenever you need to test or publish your changes to topic
   branches, merge them into "next" branch.

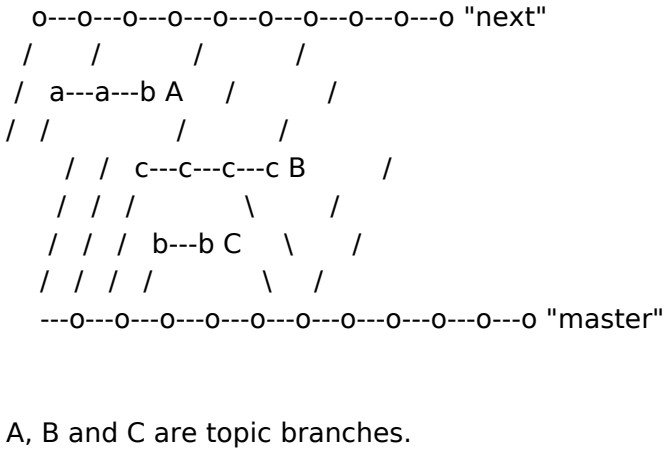
The script, being an example, hardcodes the publish branch name
to be "next", but it is trivial to make it configurable via
$GIT_DIR/config mechanism.

With this workflow, you would want to know:

(1) ... if a topic branch has ever been merged to "next". Young
    topic branches can have stupid mistakes you would rather
    clean up before publishing, and things that have not been
    merged into other branches can be easily rebased without
    affecting other people. But once it is published, you would
    not want to rewind it.

(2) ... if a topic branch has been fully merged to "master".
    Then you can delete it. More importantly, you should not
    build on top of it -- other people may already want to
    change things related to the topic as patches against your
    "master", so if you need further changes, it is better to
    fork the topic (perhaps with the same name) afresh from the
    tip of "master".

Let's look at this example:



A, B and C are topic branches.

 * A has one fix since it was merged up to "next".

 * B has finished. It has been fully merged up to "master" and "next",
   and is ready to be deleted.

 * C has not merged to "next" at all.

We would want to allow C to be rebased, refuse A, and encourage
B to be deleted.

To compute (1):

git rev-list ^master ^topic next
git rev-list ^master   next

if these match, topic has not merged in next at all.

To compute (2):

git rev-list master..topic

if this is empty, it is fully merged to "master".

DOC_END

File: .git/hooks/pre-commit.sample

```
#!/bin/sh
#
# An example hook script to verify what is about to be committed.
# Called by "git commit" with no arguments.  The hook should
# exit with non-zero status after issuing an appropriate message if
# it wants to stop the commit.
#
# To enable this hook, rename this file to "pre-commit".

if git rev-parse --verify HEAD >/dev/null 2>&1
then
against=HEAD
else
# Initial commit: diff against an empty tree object
against=$(git hash-object -t tree /dev/null)
fi

# If you want to allow non-ASCII filenames set this variable to true.
allownonascii=$(git config --type=bool hooks.allownonascii)

# Redirect output to stderr.
exec 1>&2

# Cross platform projects tend to avoid non-ASCII filenames; prevent
# them from being added to the repository. We exploit the fact that the
# printable range starts at the space character and ends with tilde.
if [ "$allownonascii" != "true" ] &&
# Note that the use of brackets around a tr range is ok here, (it's
# even required, for portability to Solaris 10's /usr/bin/tr), since
# the square bracket bytes happen to fall in the designated range.
test $(git diff --cached --name-only --diff-filter=A -z $against |
  LC_ALL=C tr -d '[ -~]\0' | wc -c) != 0
then
cat <<\EOF
Error: Attempt to add a non-ASCII file name.

This can cause problems if you want to work with people on other platforms.

To be portable it is advisable to rename the file.

If you know what you are doing you can disable this check using:

  git config hooks.allownonascii true
EOF
exit 1
fi

# If there are whitespace errors, print the offending file names and fail.
exec git diff-index --check --cached $against --
```

File: .git/hooks/applypatch-msg.sample

```
#!/bin/sh
#
# An example hook script to check the commit log message taken by
# applypatch from an e-mail message.
#
# The hook should exit with non-zero status after issuing an
# appropriate message if it wants to stop the commit.  The hook is
# allowed to edit the commit message file.
#
# To enable this hook, rename this file to "applypatch-msg".

. git-sh-setup
commitmsg="$(git rev-parse --git-path hooks/commit-msg)"
test -x "$commitmsg" && exec "$commitmsg" ${1+"$@"}
:
```

File: .git/hooks/fsmonitor-watchman.sample

```
#!/usr/bin/perl

use strict;
use warnings;
use IPC::Open2;

# An example hook script to integrate Watchman
# (https://facebook.github.io/watchman/) with git to speed up detecting
# new and modified files.
#
# The hook is passed a version (currently 2) and last update token
# formatted as a string and outputs to stdout a new update token and
# all files that have been modified since the update token. Paths must
# be relative to the root of the working tree and separated by a single NUL.
#
# To enable this hook, rename this file to "query-watchman" and set
# 'git config core.fsmonitor .git/hooks/query-watchman'
#
my ($version, $last_update_token) = @ARGV;

# Uncomment for debugging
# print STDERR "$0 $version $last_update_token\n";

# Check the hook interface version
if ($version ne 2) {
die "Unsupported query-fsmonitor hook version '$version'.\n" .
    "Falling back to scanning...\n";
}

my $git_work_tree = get_working_dir();

my $retry = 1;

my $json_pkg;
eval {
require JSON::XS;
$json_pkg = "JSON::XS";
1;
} or do {
require JSON::PP;
$json_pkg = "JSON::PP";
};

launch_watchman();

sub launch_watchman {
my $o = watchman_query();
if (is_work_tree_watched($o)) {
output_result($o->{clock}, @{$o->{files}});
}
}
```

```
sub output_result {
my ($clockid, @files) = @_;

# Uncomment for debugging watchman output
# open (my $fh, ">", ".git/watchman-output.out");
# binmode $fh, ":utf8";
# print $fh "$clockid\n@files\n";
# close $fh;

binmode STDOUT, ":utf8";
print $clockid;
print "\0";
local $, = "\0";
print @files;
}

sub watchman_clock {
my $response = qx/watchman clock "$git_work_tree"/;
die "Failed to get clock id on '$git_work_tree'.\n" .
"Falling back to scanning...\n" if $? != 0;

return $json_pkg->new->utf8->decode($response);
}

sub watchman_query {
my $pid = open2(\*CHLD_OUT, \*CHLD_IN, 'watchman -j --no-pretty')
or die "open2() failed: $!\n" .
"Falling back to scanning...\n";

# In the query expression below we're asking for names of files that
# changed since $last_update_token but not from the .git folder.
#
# To accomplish this, we're using the "since" generator to use the
# recency index to select candidate nodes and "fields" to limit the
# output to file names only. Then we're using the "expression" term to
# further constrain the results.
my $last_update_line = "";
if (substr($last_update_token, 0, 1) eq "c") {
$last_update_token = "\"$last_update_token\"";
$last_update_line = qq[\n"since": $last_update_token,];
}
my $query = <<"END";
["query", "$git_work_tree", {$last_update_line
"fields": ["name"],
"expression": ["not", ["dirname", ".git"]]
}]
END

# Uncomment for debugging the watchman query
# open (my $fh, ">", ".git/watchman-query.json");
# print $fh $query;
# close $fh;

print CHLD_IN $query;
```

```
close CHLD_IN;
my $response = do {local $/; <CHLD_OUT>};

# Uncomment for debugging the watch response
# open ($fh, ">", ".git/watchman-response.json");
# print $fh $response;
# close $fh;

die "Watchman: command returned no output.\n" .
"Falling back to scanning...\n" if $response eq "";
die "Watchman: command returned invalid output: $response\n" .
"Falling back to scanning...\n" unless $response =~ /^\{/;

return $json_pkg->new->utf8->decode($response);
}

sub is_work_tree_watched {
my ($output) = @_;
my $error = $output->{error};
if ($retry > 0 and $error and $error =~ m/unable to resolve root .* directory (.*) is not watched/) {
$retry--;
my $response = qx/watchman watch "$git_work_tree"/;
die "Failed to make watchman watch '$git_work_tree'.\n" .
    "Falling back to scanning...\n" if $? != 0;
$output = $json_pkg->new->utf8->decode($response);
$error = $output->{error};
die "Watchman: $error.\n" .
"Falling back to scanning...\n" if $error;

# Uncomment for debugging watchman output
# open (my $fh, ">", ".git/watchman-output.out");
# close $fh;

# Watchman will always return all files on the first query so
# return the fast "everything is dirty" flag to git and do the
# Watchman query just to get it over with now so we won't pay
# the cost in git to look up each individual file.
my $o = watchman_clock();
$error = $output->{error};

die "Watchman: $error.\n" .
"Falling back to scanning...\n" if $error;

output_result($o->{clock}, ("/"));
$last_update_token = $o->{clock};

eval { launch_watchman() };
return 0;
}

die "Watchman: $error.\n" .
"Falling back to scanning...\n" if $error;

return 1;
```

```
}

sub get_working_dir {
my $working_dir;
if ($^O =~ 'msys' || $^O =~ 'cygwin') {
$working_dir = Win32::GetCwd();
$working_dir =~ tr/\\/\//;
} else {
require Cwd;
$working_dir = Cwd::cwd();
}

return $working_dir;
}
```

File: .git/hooks/pre-receive.sample

```
#!/bin/sh
#
# An example hook script to make use of push options.
# The example simply echoes all push options that start with 'echoback='
# and rejects all pushes when the "reject" push option is used.
#
# To enable this hook, rename this file to "pre-receive".

if test -n "$GIT_PUSH_OPTION_COUNT"
then
i=0
while test "$i" -lt "$GIT_PUSH_OPTION_COUNT"
do
eval "value=\$GIT_PUSH_OPTION_$i"
case "$value" in
echoback=*)
echo "echo from the pre-receive-hook: ${value#*=}" >&2
;;
reject)
exit 1
esac
i=$((i + 1))
done
fi
```

File: .git/hooks/prepare-commit-msg.sample

```
#!/bin/sh
#
# An example hook script to prepare the commit log message.
# Called by "git commit" with the name of the file that has the
# commit message, followed by the description of the commit
# message's source.  The hook's purpose is to edit the commit
# message file.  If the hook fails with a non-zero status,
# the commit is aborted.
#
# To enable this hook, rename this file to "prepare-commit-msg".

# This hook includes three examples. The first one removes the
# "# Please enter the commit message..." help message.
#
# The second includes the output of "git diff --name-status -r"
# into the message, just before the "git status" output.  It is
# commented because it doesn't cope with --amend or with squashed
# commits.
#
# The third example adds a Signed-off-by line to the message, that can
# still be edited.  This is rarely a good idea.

COMMIT_MSG_FILE=$1
COMMIT_SOURCE=$2
SHA1=$3

/usr/bin/perl -i.bak -ne 'print unless(m/^. Please enter the commit message/..m/^#$/)' "$COMMIT_MSG_FILE"

# case "$COMMIT_SOURCE,$SHA1" in
#  ,|template,)
#    /usr/bin/perl -i.bak -pe '
#       print "\n" . `git diff --cached --name-status -r`
#  if /^#/ && $first++ == 0' "$COMMIT_MSG_FILE" ;;
#  *) ;;
# esac

# SOB=$(git var GIT_COMMITTER_IDENT | sed -n 's/^\(.*>\).*$/Signed-off-by: \1/p')
# git interpret-trailers --in-place --trailer "$SOB" "$COMMIT_MSG_FILE"
# if test -z "$COMMIT_SOURCE"
# then
#   /usr/bin/perl -i.bak -pe 'print "\n" if !$first_line++' "$COMMIT_MSG_FILE"
# fi
```

File: .git/hooks/post-update.sample

```
#!/bin/sh
#
# An example hook script to prepare a packed repository for use over
# dumb transports.
#
# To enable this hook, rename this file to "post-update".

exec git update-server-info
```

File: .git/hooks/pre-merge-commit.sample

```
#!/bin/sh
#
# An example hook script to verify what is about to be committed.
# Called by "git merge" with no arguments.  The hook should
# exit with non-zero status after issuing an appropriate message to
# stderr if it wants to stop the merge commit.
#
# To enable this hook, rename this file to "pre-merge-commit".

. git-sh-setup
test -x "$GIT_DIR/hooks/pre-commit" &&
        exec "$GIT_DIR/hooks/pre-commit"
:
```

File: .git/hooks/pre-applypatch.sample

```
#!/bin/sh
#
# An example hook script to verify what is about to be committed
# by applypatch from an e-mail message.
#
# The hook should exit with non-zero status after issuing an
# appropriate message if it wants to stop the commit.
#
# To enable this hook, rename this file to "pre-applypatch".

. git-sh-setup
precommit="$(git rev-parse --git-path hooks/pre-commit)"
test -x "$precommit" && exec "$precommit" ${1+"$@"}
:
```

File: .git/hooks/pre-push.sample

```
#!/bin/sh

# An example hook script to verify what is about to be pushed.  Called by "git
# push" after it has checked the remote status, but before anything has been
# pushed.  If this script exits with a non-zero status nothing will be pushed.
#
# This hook is called with the following parameters:
#
# $1 -- Name of the remote to which the push is being done
# $2 -- URL to which the push is being done
#
# If pushing without using a named remote those arguments will be equal.
#
# Information about the commits which are being pushed is supplied as lines to
# the standard input in the form:
#
#   <local ref> <local oid> <remote ref> <remote oid>
#
# This sample shows how to prevent push of commits where the log message starts
# with "WIP" (work in progress).

remote="$1"
url="$2"

zero=$(git hash-object --stdin </dev/null | tr '[0-9a-f]' '0')

while read local_ref local_oid remote_ref remote_oid
do
if test "$local_oid" = "$zero"
then
# Handle delete
:
else
if test "$remote_oid" = "$zero"
then
# New branch, examine all commits
range="$local_oid"
else
# Update to existing branch, examine new commits
range="$remote_oid..$local_oid"
fi

# Check for WIP commit
commit=$(git rev-list -n 1 --grep '^WIP' "$range")
if test -n "$commit"
then
echo >&2 "Found WIP commit in $local_ref, not pushing"
exit 1
fi
fi
done
```

```
exit 0
```

File: .git/hooks/update.sample

```
#!/bin/sh
#
# An example hook script to block unannotated tags from entering.
# Called by "git receive-pack" with arguments: refname sha1-old sha1-new
#
# To enable this hook, rename this file to "update".
#
# Config
# ------
# hooks.allowunannotated
#   This boolean sets whether unannotated tags will be allowed into the
#   repository.  By default they won't be.
# hooks.allowdeletetag
#   This boolean sets whether deleting tags will be allowed in the
#   repository.  By default they won't be.
# hooks.allowmodifytag
#   This boolean sets whether a tag may be modified after creation. By default
#   it won't be.
# hooks.allowdeletebranch
#   This boolean sets whether deleting branches will be allowed in the
#   repository.  By default they won't be.
# hooks.denycreatebranch
#   This boolean sets whether remotely creating branches will be denied
#   in the repository.  By default this is allowed.
#

# --- Command line
refname="$1"
oldrev="$2"
newrev="$3"

# --- Safety check
if [ -z "$GIT_DIR" ]; then
echo "Don't run this script from the command line." >&2
echo " (if you want, you could supply GIT_DIR then run" >&2
echo "  $0 <ref> <oldrev> <newrev>)" >&2
exit 1
fi

if [ -z "$refname" -o -z "$oldrev" -o -z "$newrev" ]; then
echo "usage: $0 <ref> <oldrev> <newrev>" >&2
exit 1
fi

# --- Config
allowunannotated=$(git config --type=bool hooks.allowunannotated)
allowdeletebranch=$(git config --type=bool hooks.allowdeletebranch)
denycreatebranch=$(git config --type=bool hooks.denycreatebranch)
allowdeletetag=$(git config --type=bool hooks.allowdeletetag)
allowmodifytag=$(git config --type=bool hooks.allowmodifytag)

# check for no description
```

```
projectdesc=$(sed -e '1q' "$GIT_DIR/description")
case "$projectdesc" in
"Unnamed repository"* | "")
echo "*** Project description file hasn't been set" >&2
exit 1
;;
esac

# --- Check types
# if $newrev is 0000...0000, it's a commit to delete a ref.
zero=$(git hash-object --stdin </dev/null | tr '[0-9a-f]' '0')
if [ "$newrev" = "$zero" ]; then
newrev_type=delete
else
newrev_type=$(git cat-file -t $newrev)
fi

case "$refname","$newrev_type" in
refs/tags/*,commit)
# un-annotated tag
short_refname=${refname##refs/tags/}
if [ "$allowunannotated" != "true" ]; then
echo "*** The un-annotated tag, $short_refname, is not allowed in this repository" >&2
echo "*** Use 'git tag [ -a | -s ]' for tags you want to propagate." >&2
exit 1
fi
;;
refs/tags/*,delete)
# delete tag
if [ "$allowdeletetag" != "true" ]; then
echo "*** Deleting a tag is not allowed in this repository" >&2
exit 1
fi
;;
refs/tags/*,tag)
# annotated tag
if [ "$allowmodifytag" != "true" ] && git rev-parse $refname > /dev/null 2>&1
then
echo "*** Tag '$refname' already exists." >&2
echo "*** Modifying a tag is not allowed in this repository." >&2
exit 1
fi
;;
refs/heads/*,commit)
# branch
if [ "$oldrev" = "$zero" -a "$denycreatebranch" = "true" ]; then
echo "*** Creating a branch is not allowed in this repository" >&2
exit 1
fi
;;
refs/heads/*,delete)
# delete branch
if [ "$allowdeletebranch" != "true" ]; then
echo "*** Deleting a branch is not allowed in this repository" >&2
```

```
exit 1
fi
;;
refs/remotes/*,commit)
# tracking branch
;;
refs/remotes/*,delete)
# delete tracking branch
if [ "$allowdeletebranch" != "true" ]; then
echo "*** Deleting a tracking branch is not allowed in this repository" >&2
exit 1
fi
;;
*)
# Anything else (is there anything else?)
echo "*** Update hook: unknown type of update to ref $refname of type $newrev_type" >&2
exit 1
;;
esac

# --- Finished
exit 0
```

File: .git/hooks/push-to-checkout.sample

```
#!/bin/sh

# An example hook script to update a checked-out tree on a git push.
#
# This hook is invoked by git-receive-pack(1) when it reacts to git
# push and updates reference(s) in its repository, and when the push
# tries to update the branch that is currently checked out and the
# receive.denyCurrentBranch configuration variable is set to
# updateInstead.
#
# By default, such a push is refused if the working tree and the index
# of the remote repository has any difference from the currently
# checked out commit; when both the working tree and the index match
# the current commit, they are updated to match the newly pushed tip
# of the branch. This hook is to be used to override the default
# behaviour; however the code below reimplements the default behaviour
# as a starting point for convenient modification.
#
# The hook receives the commit with which the tip of the current
# branch is going to be updated:
commit=$1

# It can exit with a non-zero status to refuse the push (when it does
# so, it must not modify the index or the working tree).
die () {
echo >&2 "$*"
exit 1
}

# Or it can make any necessary changes to the working tree and to the
# index to bring them to the desired state when the tip of the current
# branch is updated to the new commit, and exit with a zero status.
#
# For example, the hook can simply run git read-tree -u -m HEAD "$1"
# in order to emulate git fetch that is run in the reverse direction
# with git push, as the two-tree form of git read-tree -u -m is
# essentially the same as git switch or git checkout that switches
# branches while keeping the local changes in the working tree that do
# not interfere with the difference between the branches.

# The below is a more-or-less exact translation to shell of the C code
# for the default behaviour for git's push-to-checkout hook defined in
# the push_to_deploy() function in builtin/receive-pack.c.
#
# Note that the hook will be executed from the repository directory,
# not from the working tree, so if you want to perform operations on
# the working tree, you will have to adapt your code accordingly, e.g.
# by adding "cd .." or using relative paths.

if ! git update-index -q --ignore-submodules --refresh
then
die "Up-to-date check failed"
```

```
fi

if ! git diff-files --quiet --ignore-submodules --
then
die "Working directory has unstaged changes"
fi

# This is a rough translation of:
#
#   head_has_history() ? "HEAD" : EMPTY_TREE_SHA1_HEX
if git cat-file -e HEAD 2>/dev/null
then
head=HEAD
else
head=$(git hash-object -t tree --stdin </dev/null)
fi

if ! git diff-index --quiet --cached --ignore-submodules $head --
then
die "Working directory has staged changes"
fi

if ! git read-tree -u -m "$commit"
then
die "Could not update working tree to new HEAD"
fi
```

File: .git/refs/heads/main

cdd15676887f405925ed6289b0515de8c8c96f6d

File: .git/refs/remotes/origin/main

cdd15676887f405925ed6289b0515de8c8c96f6d

File: repo2pdf/__init__.py

```
# __init__.py

__version__ = '0.1.0'
```

File: repo2pdf/core.py

```
import os
import shutil
from git import Repo
from repo2pdf.pdf import generate_pdf
import pathspec
from repo2pdf.utils import output_json

def process_local_repo(path, want_json=False, output_path=None, exclude_list=None):
    print(f"Processing local repo at {path}...")
    files = traverse_repo(path, exclude_list or [])
    output_path = output_path or os.path.join(os.getcwd(), "repo_output.pdf")
    generate_pdf(files, output_path)
    print(f"PDF saved to {output_path}")

    if want_json:
        output_json(files, output_path)

def process_remote_repo(url, want_json=False, output_path=None, exclude_list=None, delete=True):
    tmp_dir = "./tmp_repo"
    print(f"Cloning {url} into {tmp_dir}...")
    Repo.clone_from(url, tmp_dir)
    files = traverse_repo(tmp_dir, exclude_list or [])
    output_path = output_path or os.path.join(os.getcwd(), "repo_output.pdf")
    generate_pdf(files, output_path)
    print(f"PDF saved to {output_path}")

    if want_json:
        output_json(files, output_path)

    if delete and os.path.exists(tmp_dir):
        shutil.rmtree(tmp_dir)
        print("Temporary repo deleted")


def traverse_repo(path, exclude_list=[]):
    # Load .gitignore
    gitignore_path = os.path.join(path, '.gitignore')
    spec = None
    if os.path.exists(gitignore_path):
        with open(gitignore_path) as f:
            spec = pathspec.PathSpec.from_lines('gitwildmatch', f)

    file_data = []
    for root, dirs, files in os.walk(path):
        for file in files:
            # Skip excluded extensions
            if any(file.endswith(ext) for ext in exclude_list):
                continue

            file_path = os.path.join(root, file)
            relative_path = os.path.relpath(file_path, path)
```

```
        # Skip ignored files
        if spec and spec.match_file(relative_path):
            continue

        try:
            with open(file_path, 'r', encoding='utf-8') as f:
                content = f.read()
            file_data.append((relative_path, content))
        except Exception as e:
            print(f"Skipping {file_path}: {e}")
return file_data
```

File: repo2pdf/pdf.py

```
import os
from fpdf import FPDF

def generate_pdf(files, output_path):
    pdf = FPDF()
    pdf.set_auto_page_break(auto=True, margin=15)

    # Use bundled font
    font_path = os.path.join(os.path.dirname(__file__), "fonts", "DejaVuSans.ttf")
    pdf.add_font("DejaVu", "", font_path)
    pdf.set_font("DejaVu", size=10)

    for filename, content in files:
        pdf.add_page()
        pdf.multi_cell(0, 5, f"File: {filename}\n\n{content}")

    pdf.output(output_path)
```

File: repo2pdf/cli.py

```
import inquirer
from repo2pdf.core import process_local_repo, process_remote_repo

def main():
    ascii_art = r"""
 ______   ______  ______  ______          _____          ______  ______  ______
/_____/\ /_____/\/_____/\/_____/\         /_____/\        /_____/\/_____/\/_____/\
\:::_ \ \\::::_\/\:::_ \ \\:::_ \ \  ______\:::_:\ \ ______\:::_ \ \:::_ \ \::::_\/_
 \:(_) ) )\:\/___/\:(_) \ \\:\ \ \ \/______/\  _\:\|/______/\:(_) \ \:\ \ \ \:\/___/\
  \: __ `\ \::___\/\: ___\/\:\ \ \ \__::::\/ /::_/_\__::::\/\: ___\/\:\ \ \ \::._\/
   \ \ `\ \ \:\____/\ \ \   \:\_\ \ \        \:\____/\       \ \ \   \:\/.:| \:\ \
    \_\/ \_\/\_____\/\_\/    \_____\/         \_____\/        \_\/    \____/_/\_\/


Welcome to repo-pdf – convert your repositories to PDFs

Built by Haris

    """
    print(ascii_art)

    repo_type_q = [
        inquirer.List('repo_type',
                      message="Do you want to generate a PDF from a local or remote repo?",
                      choices=['Local', 'Remote'])
    ]
    repo_type = inquirer.prompt(repo_type_q)['repo_type']

    json_q = [
        inquirer.Confirm('json', message="Do you also want to generate a JSON version?", default=True)
    ]
    want_json = inquirer.prompt(json_q)['json']

    output_q = [
        inquirer.Text('output', message="Provide output path for PDF (press enter for current directory)")
    ]
    output_path = inquirer.prompt(output_q)['output']

    exclude_q = [
          inquirer.Text('exclude', message="Enter file extensions to exclude (e.g. .png,.jpg,.exe), or press
enter to skip")
    ]
    exclude_input = inquirer.prompt(exclude_q)['exclude']
    exclude_list = [e.strip() for e in exclude_input.split(',')] if exclude_input else []

    if repo_type == 'Local':
        path_q = [
            inquirer.Text('path', message="Provide local repo path (or press enter if in root)")
        ]
        path = inquirer.prompt(path_q)['path']
        process_local_repo(path or '.', want_json, output_path, exclude_list)
```

```
    else:
        url_q = [
            inquirer.Text('url', message="Provide GitHub repo URL")
        ]
        url = inquirer.prompt(url_q)['url']

        delete_q = [
                    inquirer.Confirm('delete', message="Do you want to delete the cloned repo after PDF
generation?", default=True)
        ]
        delete = inquirer.prompt(delete_q)['delete']

        process_remote_repo(url, want_json, output_path, exclude_list, delete)

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

File: repo2pdf/utils.py

```
import os
import mimetypes
import json

EXTENSION_LANGUAGE_MAP = {
    # Programming languages
    '.py': 'Python',
    '.js': 'JavaScript',
    '.ts': 'TypeScript',
    '.java': 'Java',
    '.c': 'C',
    '.cpp': 'C++',
    '.cs': 'C#',
    '.rb': 'Ruby',
    '.go': 'Go',
    '.rs': 'Rust',
    '.php': 'PHP',
    '.swift': 'Swift',
    '.kt': 'Kotlin',
    '.m': 'Objective-C',
    '.scala': 'Scala',
    '.sh': 'Shell Script',
    '.bat': 'Batch Script',
    '.ps1': 'PowerShell',
    '.pl': 'Perl',
    '.r': 'R',

    # Web & markup
    '.html': 'HTML',
    '.htm': 'HTML',
    '.css': 'CSS',
    '.scss': 'SCSS',
    '.sass': 'SASS',
    '.less': 'LESS',
    '.json': 'JSON',
    '.xml': 'XML',
    '.yml': 'YAML',
    '.yaml': 'YAML',
    '.md': 'Markdown',

    # Config & data
    '.env': 'Environment Config',
    '.ini': 'INI Config',
    '.conf': 'Config',
    '.cfg': 'Config',
    '.toml': 'TOML Config',
    '.gradle': 'Gradle Build File',
    '.dockerfile': 'Dockerfile',

    # Text & miscellaneous
    '.txt': 'Plain Text',
    '.log': 'Log File',
```

```
    '.csv': 'CSV',
    '.tsv': 'TSV',
}


def output_json(files, output_path):
    data = []
    for filename, content in files:
        ext = os.path.splitext(filename)[1]
        language = EXTENSION_LANGUAGE_MAP.get(ext)

        if not language:
            # Fall back to mimetypes
            mime_type, _ = mimetypes.guess_type(filename)
            if mime_type:
                # Use the subtype (e.g. 'plain' from 'text/plain') or mime_type as fallback
                language = mime_type.split('/')[1] if '/' in mime_type else mime_type
            else:
                language = 'Unknown'

        data.append({
            "path": filename,
            "language": language,
            "content": content
        })

    json_path = output_path.replace(".pdf", ".json")
    with open(json_path, 'w') as f:
        json.dump({"files": data}, f, indent=2)

    print(f" JSON saved to {json_path}")
```