

git2pdf v0.1.0

git2pdf

Version: 0.1.0

Convert git repositories to PDF for code review

Files: 0

```
200 //! Rust crate discovery in a repository

use std::fs;
use std::path::{Path, PathBuf};

use anyhow::{Context, Result};
use serde::Deserialize;

/// Information about a discovered Rust crate
#[derive(Debug, Clone)]
pub struct CrateInfo {
    /// Crate name from Cargo.toml
    pub name: String,
    /// Path to the crate root (directory containing Cargo.toml)
    pub path: PathBuf,
    /// Whether this is a workspace member
    pub is_workspace_member: bool,
    /// Crate version
    pub version: String,
    /// Crate description
    pub description: Option<String>,
}
```

```

/// Minimal Cargo.toml structure for parsing
#[derive(Debug, Deserialize)]
struct CargoToml {
    package: Option<Package>,
    workspace: Option<Workspace>,
}

#[derive(Debug, Deserialize)]
struct Package {
    name: String,
    #[serde(default = "default_version")]
    version: String,
    description: Option<String>,
}

fn default_version() -> String {
    "0.0.0".to_string()
}

#[derive(Debug, Deserialize)]
struct Workspace {
    members: Option<Vec<String>>,
    #[serde(default)]
    exclude: Vec<String>,
}

/// Discover all Rust crates in a repository

```

```

pub fn
discover_crates(repo_path:&Path) -> Result<Vec<CrateInfo>>; {
    let mut crates = Vec::new();

```

```

// Check if there's a root Cargo.toml
let root_cargo = repo_path.join("Cargo.toml");

if !root_cargo.exists(){
// No Cargo.toml at root, search recursively
return discover_crates_recursive(repo_path);
}

let content = fs::read_to_string(&root_cargo)
    .context("Failed to read root Cargo.toml")?;

let cargo_toml: CargoToml = toml::from_str(&content)
    .context("Failed to parse root Cargo.toml")?;

// Check if it's a workspace
if let Some(workspace) = cargo_toml.workspace {
// It's a workspace - discover members
if let Some(members) = workspace.members {
for member_pattern in members {
let member_crates
=
e
x
p
a
n
d_workspace_member(repo_path, &member_pattern, &workspace.exclude)?;
crates.extend(member_crates);
}
}

// Also check if the root is a package
if let Some(package) = cargo_toml.package {
crates.push(CrateInfo {
    name: package.name,
    path: repo_path.to_path_buf(),
    is_workspace_member: false,
    version: package.version,

```

```

        description: package.description,
    });
}
}elseif let Some(package) = cargo_toml.package {
    // It's a single crate
    crates.push(CrateInfo {
        name: package.name,
        path: repo_path.to_path_buf(),
        is_workspace_member: false,
        version: package.version,
        description: package.description,
    });
}

// Sort by name for consistent output
crates.sort_by(|a, b| a.name.cmp(&b.name));

// Remove duplicates (by path)
crates.dedup_by(|a, b| a.path == b.path);

Ok(crates)
}

/// Expand a workspace member pattern (supports glob patterns like
/// "crates/*")
fn expand_workspace_member(
    repo_path: &Path,
    pattern: &str,
    exclude: &[String],
) -> Result<Vec<CrateInfo>> {
    let mut crates = Vec::new();

    if pattern.contains('*&#39;*&#39;){
        // It's a glob pattern
        let base_path =
            repo_path.join(pattern.split('*&#39;*&#39;).next().unwrap_or(&quot;&quot;));

```

```

if base_path.exists() && base_path.is_dir(){
  for entry in fs::read_dir(&base_path)?{
    let entry = entry?;
    let path = entry.path();

    if path.is_dir(){
      // Check if excluded
      let rel_path = path.strip_prefix(repo_path)
        .map(|p| p.to_string_lossy().to_string())
        .unwrap_or_default();

      if exclude.iter().any(|e| rel_path.starts_with(e)){
        continue;
      }

      if let Some(crate_info) = try_parse_crate(&path)?{
        crates.push(CrateInfo {
          is_workspace_member: true,
          ..crate_info
        });
      }
    }
  }
} else {
  // Exact path
  let member_path = repo_path.join(pattern);

  // Check if excluded
  if exclude.iter().any(|e| pattern.starts_with(e)){
    return Ok(crates);
  }

  if let Some(crate_info) = try_parse_crate(&member_path)?{
    crates.push(CrateInfo {

```

```

        is_workspace_member: true,

        ..crate_info
    });
}
}

Ok(crates)
}

/// Try to parse a crate from a directory

fn
n
try_parse_crate(path: &Path) -> Result<Option<CrateInfo>>; {
    let cargo_path = path.join(&quot;Cargo.toml&quot;);

    if !cargo_path.exists() {
        return Ok(None);
    }

    let content = fs::read_to_string(&cargo_path)
        .context(&quot;Failed to read Cargo.toml&quot;)?;

    let cargo_toml: CargoToml = toml::from_str(&content)
        .context(&quot;Failed to parse Cargo.toml&quot;)?;

    if let Some(package) = cargo_toml.package {
        Ok(Some(CrateInfo {
            name: package.name,
            path: path.to_path_buf(),
            is_workspace_member: false,
            version: package.version,
            description: package.description,
        }))
    } else {
        Ok(None)
    }
}

```

```
}
```

```
    /// Recursively discover crates when there's no workspace,
    respecting .gitignore
```

```
fn
discover
crates_recursive(repo_path:&Path)->Result<Vec<CrateInfo>>,{
    useignore::WalkBuilder;

    letmut crates =Vec::new();

    let walker =WalkBuilder::new(repo_path)
        .hidden(true)// Skip hidden files/directories
        .git_ignore(true)// Respect .gitignore
        .git_global(true)// Respect global gitignore
        .git_exclude(true)// Respect .git/info/exclude
        .parents(true)// Check parent directories for ignore files
        .follow_links(false)
        .build();

    for entry in walker {
        let entry = entry?;
        let path = entry.path();

        // Skip target and node_modules explicitly
        if path.components().any(|c|{
            let s = c.as_os_str().to_string_lossy();
            s =="target"|| s =="node_modules"
        }){
```

```

        continue;
    }

    if path.file_name().map(|n|
==&quot;Cargo.toml&quot;).unwrap_or(false){
        let parent = path.parent().unwrap_or(repo_path);
        if let Some(crate_info) = try_parse_crate(parent)? {
            crates.push(crate_info);
        }
    }
}

// Sort by name
crates.sort_by(|a, b| a.name.cmp(&b.name));

Ok(crates)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_discover_crates_single() {
        // This would need a test fixture
    }
}

```

```

200 //! File classification for Rust projects
    //!
    //! Classifies files as source code, tests, integration tests,
examples, etc.
    //! Respects .gitignore files using the `ignore` crate.

    use std::fs;
    use std::path::{Path, PathBuf};

```

```

use anyhow::Result;
use ignore::WalkBuilder;

/// Category of a source file
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum FileCategory {
    /// Main source code (src/)
    Source,
    /// Unit tests (inline #[cfg(test)] or tests/ inside src/)
    Test,
    /// Integration tests (tests/ at crate root)
    IntegrationTest,
    /// Examples (examples/)
    Example,
    /// Benchmarks (benches/)
    Benchmark,
    /// Build script
    BuildScript,
    /// Other Rust files
    Other,
}

/// A classified source file
#[derive(Debug, Clone)]
pub struct SourceFile {
    /// Absolute path to the file
    pub path: PathBuf,
    /// Path relative to crate root
    pub relative_path: PathBuf,
    /// File category
    pub category: FileCategory,
    /// Module path (e.g., "crate::foo::bar");

```

```

pubmodule_path: String,
}

/// Classify all Rust files in a crate, respecting .gitignore
pubfnclassify_files(crate_path:&Path,
include_tests:bool)->Result<Vec<SourceFile>>{
    letmut files =Vec::new();

    // Use ignore crate's WalkBuilder which respects .gitignore
    let walker =WalkBuilder::new(crate_path)
        .hidden(true)// Skip hidden files/directories
        .git_ignore(true)// Respect .gitignore
        .git_global(true)// Respect global gitignore
        .git_exclude(true)// Respect .git/info/exclude
        .parents(true)// Check parent directories for ignore files
        .follow_links(false)
        .build();

    for entry in walker {
        let entry = entry?;
        let path = entry.path();

        // Skip directories
        if!path.is_file(){
            continue;
        }

        // Only process Rust files
        if path.extension().map(|ele|ele !="rs").unwrap_or(true){
            continue;
        }

        // Skip target directory explicitly (in case it's not in
        .gitignore)
        let relative_path = path.strip_prefix(crate_path)
            .unwrap_or(path)
            .to_path_buf();
    }

```

```

    if
relative_path.components().any(|c|c.as_os_str()==&quot;target&quot;){
    continue;
}

let category =classify_file(&relative_path);

// Skip tests if not included
if!include_tests &&matches!(category,FileCategory::Test
|FileCategory::IntegrationTest){
    continue;
}

let module_path =compute_module_path(&relative_path);

    files.push(SourceFile {
        path: path.to_path_buf(),
        relative_path,
        category,
        module_path,
    });
}

// Sort files by their path for consistent ordering
files.sort_by(|a,b|a.relative_path.cmp(&b.relative_path));

Ok(files)
}

/// Classify a file based on its relative path
fnclassify_file(relative_path:&Path)-&gt; FileCategory{
let components:Vec<_>= relative_path.components()
    .map(|c|c.as_os_str().to_string_lossy().to_string())
    .collect();

if components.is_empty(){
returnFileCategory::Other;
}

```

```

let first = &components[0];
let file_name = relative_path.file_name()
    .map(|ln|ln.to_string_lossy().to_string())
    .unwrap_or_default();

// Check for build.rs at root
if components.len() == 1 && file_name == "build.rs" {
    return FileCategory::BuildScript;
}

// Check top-level directory
match first.as_str(){
    "src" => {
        // Check if it's in a tests subdirectory inside src
        if components.iter().any(|cl|cl == "tests"){
            FileCategory::Test
        }else{
            FileCategory::Source
        }
    }
    "tests" => FileCategory::IntegrationTest,
    "examples" => FileCategory::Example,
    "benches" => FileCategory::Benchmark,
    _ => FileCategory::Other,
}

}

/// Compute the module path for a file
fn compute_module_path(relative_path:&Path) -> String{
    let components:Vec<_>= relative_path.components()
        .map(|cl|cl.as_os_str().to_string_lossy().to_string())
        .collect();

```

```

if components.is_empty(){
    return"crate".to_string();
}

letmut path_parts =Vec::new();
    path_parts.push("crate".to_string());

// Skip the first component if it's "src"
let start_idx =if
components.first().map(|s|s.as_str())==Some("src"){1}else{0};

for(idx, component)in components[start_idx..].iter().enumerate(){
// Remove .rs extension from the last component
let name =if idx == components.len()- start_idx -1{
    component.trim_end_matches(".rs");
}else{
    component
};

// Handle mod.rs and lib.rs specially
if name =="mod"|| name =="lib"|| name
=="main"{
    continue;
}

    path_parts.push(name.to_string());
}

    path_parts.join("::");
}

/// Check if a file contains test code (has #[test] or
#[cfg(test)])
pubfnfile_contains_tests(path:&Path)->Result<bool>{
    let content =fs::read_to_string(path)?;

    Ok(content.contains("#[test]")||
content.contains("#[cfg(test)]"))
}

```

```
#[cfg(test)]
```

```
mod tests {
```

```
    use super::*;
```

```
#[test]
```

```
fn test_classify_source() {
```

```
    assert!(
        classify_file(Path::new("src/lib.rs")), FileCategory::Source);
```

```
    assert!(
        classify_file(Path::new("src/foo/mod.rs")), FileCategory::Source);
```

```
    assert!(
        classify_file(Path::new("src/bar.rs")), FileCategory::Source);
}
```

```
#[test]
```

```
fn test_classify_tests() {
```

```
    assert
```

```

t
-
e
q
!
(
c
l
a
s
s
i
f
y
-
f
i
l
e
(
P
a
th::new(&quot;tests/integration.rs&quot;)),FileCategory::IntegrationTest);

a
s
s
e
r
t
-
e
q
!
(
c
l
assify_file(Path::new(&quot;src/tests/unit.rs&quot;)),FileCategory::Test);
    }

    #[test]
    fn test_classify_examples(){

a
s
s
e
r
t
-
e
q

```

```

!
(
c
l
a
s
sify_file(Path::new("examples/demo.rs")),FileCategory::Example);
    }

```

```

    #[test]
    fn test_module_path(){

```

```

a
s
s
e
r
t
-
e
q
!
(
compute_module_path(Path::new("src/lib.rs")),"crate");

```

```

a
s
s
e
r
t
-
e
q
!
(
c
o
m
p
u
te_module_path(Path::new("src/foo.rs")),"crate::foo");

```

```

a
s
s
e
r
t
-
e
q
!

```

```
(
c
o
m
p
u
t
e
-
module_path(Path::new("src/foo/mod.rs")),"crate::foo");

a
s
s
e
r
t
-
eq!
(
c
o
m
p
u
t
e
-
module_path(Path::new("src/foo/bar.rs")),"crate::foo::bar");
    }
}
```

```
100 //! Git operations using gitoxide (gix)
```

```
use std::path::Path;
```

```
use anyhow::{Context, Result, bail};
```

```
/// Clone a repository or open it if it already exists
```

```
pub fn clone_or_open_repo(url:&str, dest:&Path,
```

```

verbose:bool)->Result<()>{
    if dest.exists()&& dest.join(".git").exists(){
        if verbose {
            println!("Repository already exists at {}",
dest.display());
        }

        // Optionally fetch latest changes
        if let Err(e)=fetch_repo(dest, verbose){
            if verbose {
                println!("Warning: Could not fetch latest changes: {}",
e);
            }
        }

        return Ok(());
    }

    if dest.exists(){
        std::fs::remove_dir_all(dest)
            .context("Failed to remove existing
directory")?;
    }

    if verbose {
        println!("Cloning repository from {}...", url);
    }

    // Prepare clone using gix
    let url =gix::url::parse(url.into())
        .context("Failed to parse git URL")?;

    let mut prepare =gix::prepare_clone(url, dest)
        .context("Failed to prepare clone")?;

    // Perform the fetch
    let(mut checkout, _outcome)= prepare

```

```

f
e
t
c
h
.then_checkout(gix::progress::Discard,&gix::interrupt::IS_INTERRUPTED)
    .context(&quot;Failed to fetch repository&quot;)?;

// Checkout the main worktree
let(_repo, _outcome)= checkout

.main_worktree(gix::progress::Discard,&gix::interrupt::IS_INTERRUPTED)
    .context(&quot;Failed to checkout worktree&quot;)?;

if verbose {
println!(&quot;Clone complete&quot;);
}

Ok(())
}

/// Fetch the latest changes from the remote
fnfetch_repo(repo_path:&Path,
verbose:bool)-&gt;Result<()&gt;{
    if verbose {
println!(&quot;Fetching latest changes...&quot;);
    }

    let repo =gix::open(repo_path)
        .context(&quot;Failed to open repository&quot;)?;

    let remote =
repo.find_default_remote(gix::remote::Direction::Fetch)
        .context(&quot;No default remote found&quot;)?
        .context(&quot;Failed to find remote&quot;)?;

    let _outcome = remote
        .connect(gix::remote::Direction::Fetch)
        .context(&quot;Failed to connect to remote&quot;)?
        .prepare_fetch(gix::progress::Discard,Default::default())
        .context(&quot;Failed to prepare fetch&quot;)?

```

```

.receive(gix::progress::Discard,&gix::interrupt::IS_INTERRUPTED)
    .context(&quot;Failed to fetch&quot;)?;

    if verbose {
        println!(&quot;Fetch complete&quot;);
    }

    Ok(())
}

/// Checkout a specific branch, tag, or commit
pubfn checkout_ref(repo_path:&Path, git_ref:&str,
verbose:bool)-&gt;Result<()>{
    let repo =gix::open(repo_path)
        .context(&quot;Failed to open repository&quot;)?;

    // Try to find the reference
    let reference =find_reference(&repo, git_ref)?;

    if verbose {
        println!(&quot;Found reference: {}&quot;, git_ref);
    }

    // Get the commit id - peel to the actual commit
    let commit_id = reference.id().detach();

    // Update HEAD to point to this commit
    let head_ref = repo.find_reference(&quot;HEAD&quot;).ok();

    if verbose {
        println!(&quot;Checked out {} ({})&quot;, git_ref, commit_id);
    }

    Ok(())
}

/// Find a reference by name (branch, tag, or commit)
fn find_reference<A>(repo:&A,gix::Repository,
```

```

name:&str)->Result<gix::Reference,&#39;a>>{
    // Try as a local branch first

    let branch_ref =format!("refs/heads/{}", name);
    ifletOk(reference)= repo.find_reference(&branch_ref){
        returnOk(reference);
    }

    // Try as a remote branch

    let remote_ref =format!("refs/remotes/origin/{}", name);
    ifletOk(reference)= repo.find_reference(&remote_ref){
        returnOk(reference);
    }

    // Try as a tag

    let tag_ref =format!("refs/tags/{}", name);
    ifletOk(reference)= repo.find_reference(&tag_ref){
        returnOk(reference);
    }

    // Try as a full reference

    ifletOk(reference)= repo.find_reference(name){
        returnOk(reference);
    }

    bail!("Could not find reference: {}", name)
}

/// Try to checkout main or master branch
#[allow(dead_code)]
pubfncheckout_default_branch(repo_path:&Path,
verbose:bool)->Result<String>{
    let repo =gix::open(repo_path)
        .context("Failed to open repository");

    // Try &#39;main&#39; first

    iffind_reference(&repo,"main").is_ok(){

```

```

checkout_ref(repo_path,&quot;main&quot;; verbose)?;
returnOk(&quot;main&quot;.to_string());
}

// Try &#39;master&#39;
iffind_reference(&amp;repo,&quot;master&quot;).is_ok(){
checkout_ref(repo_path,&quot;master&quot;; verbose)?;
returnOk(&quot;master&quot;.to_string());
}

// Use HEAD
Ok(&quot;HEAD&quot;.to_string())
}

```

```

200 //! HTML generation with syntax highlighting
    //!
    //! Generates HTML from source files using syntect for syntax
highlighting.

```

```

usestd::fs;

useanyhow::{Context,Result};
usesyntect::highlighting::{Theme, Style, FontStyle};
usesyntect::parsing::SyntaxSet;
usesyntect::easy::HighlightLines;
usesyntect::util::LinesWithEndings;

usecrate::crate_discovery::CrateInfo;
usecrate::file_classifier::SourceFile;

/// Generate HTML for an entire crate
pubfngenerate_html_for_crate(
crate_info:&amp;CrateInfo,
files:&amp;[&amp;SourceFile],
syntax_set:&amp;SyntaxSet,

```

```

theme:&Theme,
font_size:f32,
columns:u32,
)-&Result<String>{
    letmut html =String::new();

    // HTML header with CSS
    html.push_str(&generate_html_header(crate_info, font_size,
columns, theme));

    // Generate content for each file
    for file in files {
        let file_html =generate_html_for_file(file, syntax_set, theme)?;
        html.push_str(&file_html);
    }

    // Close HTML

html.push_str("</div>\n</body>\n</html>");
    Ok(html)
}

/// Generate HTML header with CSS styling
fngenerate_html_header(crate_info:&CrateInfo, font_size:f32,
columns:u32, theme:&Theme)-&String{
    let bg_color = theme.settings.background
        .map(|c|format!("{:02x}{:02x}{:02x}", c.r, c.g,
c.b))
        .unwrap_or_else(||&"#ffffff".to_string());

    let fg_color = theme.settings.foreground
        .map(|c|format!("{:02x}{:02x}{:02x}", c.r, c.g,
c.b))
        .unwrap_or_else(||&"#000000".to_string());

    format!(r#"<!DOCTYPE html>
<html lang=en>
<head>
    <meta charset=UTF-8>

```

```
<meta name="viewport"
content="width=device-width, initial-scale=1.0">
<title>{name} - Code Review</title>
<style>

    @page {{
        size: A4;
        margin: 10mm;
    }}

    * {{
        box-sizing: border-box;
    }}

    body {{
        font-family: 'Fira Code', 'Source Code
Pro', 'Consolas', 'Monaco', monospace;
        font-size: {font_size}pt;
        line-height: 1.3;
        margin: 0;
        padding: 0;
        background-color: {bg_color};
        color: {fg_color};
    }}

    .content {{
        column-count: {columns};
        column-gap: 15px;
        column-rule: 1px solid #ddd;
        padding: 10px;
    }}

    .file-section {{
        break-inside: avoid-column;
        margin-bottom: 20px;
        page-break-inside: avoid;
    }}
```

```
.file-header {{  
    background-color: #f0f0f0;  
    color: #333;  
    padding: 5px 10px;  
    font-weight: bold;  
    font-size: {header_size}pt;  
    border-bottom: 2px solid #666;  
    margin-bottom: 5px;  
    break-after: avoid;  
}}
```

```
.file-path {{  
    font-size: {path_size}pt;  
    color: #666;  
    font-weight: normal;  
    margin-left: 10px;  
}}
```

```
.code-block {{  
    margin: 0;  
    padding: 5px;  
    overflow-x: hidden;  
    white-space: pre-wrap;  
    word-wrap: break-word;  
    font-size: {font_size}pt;  
    line-height: 1.2;  
}}
```

```
.line {{  
    display: block;  
}}
```

```
.line-number {{  
    display: inline-block;
```

```
width: 3em;
text-align: right;
padding-right: 1em;
color: #999;
user-select: none;
font-size: {line_num_size}pt;
}}
```

```
.line-content {{
  display: inline;
}}
```

```
h1 {{
  font-size: 16pt;
  margin: 10px 0;
  padding: 10px;
  background-color: #333;
  color: white;
  column-span: all;
}}
```

```
.crate-info {{
  column-span: all;
  padding: 10px;
  background-color: #f5f5f5;
  margin-bottom: 20px;
  border-left: 4px solid #333;
}}
```

```
.crate-info h2 {{
  margin: 0 0 5px 0;
  font-size: 14pt;
}}
```

```
.crate-info p {{
```

```

        margin: 5px 0;
        font-size: 10pt;
        color: #666;
    }}
    </style>
</head>
<body>
    <h1>{name} v{version}</h1>
    <div class="crate-info">
        <h2>{name}</h2>
        <p>Version: {version}</p>
        {description}
        <p>Files: {file_count}</p>
    </div>
    <div class="content">
        <#,
            name =html_escape(&crate_info.name),
            version =html_escape(&crate_info.version),
            description = crate_info.description.as_ref()

.map(|d| format!("<p>{}</p>",html_escape(d)))
        .unwrap_or_default(),
        file_count =0, // Will be updated
        font_size = font_size,
        header_size = font_size +2.0,
        path_size = font_size -1.0,
        line_num_size = font_size -1.0,
        columns = columns,
        bg_color = bg_color,
        fg_color = fg_color,
    )
}

```

```

/// Generate HTML for a single source file
fn generate_html_for_file(
    file:&SourceFile,
    syntax_set:&SyntaxSet,
    theme:&Theme,
)-&Result<String>{
    let content = fs::read_to_string(&file.path)
        .with_context(||format!("Failed to read file:
{}";, file.path.display()))?;

    // Get syntax for Rust
    let syntax = syntax_set.find_syntax_by_extension("rs");
        .unwrap_or_else(||syntax_set.find_syntax_plain_text());

    letmut highlighter = HighlightLines::new(syntax, theme);

    letmut html = String::new();

    // File section
    html.push_str(&format!(
r#"<div class="file-section">
    <div class="file-header">
    {}
        <span class="file-path">{}</span>
    </div>
    <pre class="code-block">{}</pre>
html_escape(&file.module_path),
html_escape(&file.relative_path.to_string_lossy()),
));

    // Highlight each line
    for(line_num,
line)in LinesWithEndings::from(&content).enumerate(){
        let highlighted = highlighter.highlight_line(line, syntax_set)
            .unwrap_or_else(|_|vec![(Style::default(), line)]);

        html.push_str(&format!(

```

```

        r#"<span class="line"><span
class="line-number">{}</span><span
class="line-content">>#",
            line_num + 1
    ));

    for(style, text) in highlighted {
        let css = style_to_css(&style);
        if css.is_empty(){
            html.push_str(&html_escape(text));
        }else{
            html.push_str(&format!(
                r#"<span style="{}">{}</span><span
                css,
                html_escape(text)
            ));
        }
    }

    html.push_str("</span></span>");

    html.push_str("</pre>\n</div>\n");

    Ok(html)
}

/// Convert a syntect Style to CSS
fn style_to_css(style:&Style)-> String{
    let mut css_parts = Vec::new();

    // Foreground color
    let fg = style.foreground;
    if fg.a > 0{
        css_parts.push(format!("color:
#{:02x}#{:02x}#{:02x}&quot;;", fg.r, fg.g, fg.b));
    }

```

```

// Font style
if style.font_style.contains(FontStyle::BOLD){
    css_parts.push(&quot;font-weight: bold&quot;;.to_string());
}
if style.font_style.contains(FontStyle::ITALIC){
    css_parts.push(&quot;font-style: italic&quot;;.to_string());
}
if style.font_style.contains(FontStyle::UNDERLINE){
    css_parts.push(&quot;text-decoration:
underline&quot;;.to_string());
}

    css_parts.join(&quot;; &quot;);
}

/// Escape HTML special characters
fnhtml_escape(s:&str)-&gt; String{
    s.replace(&#39;&#39;, &quot;&quot;);
    .replace(&#39;&lt;&#39;, &quot;&lt;&quot;);
    .replace(&#39;&gt;&#39;, &quot;&gt;&quot;);
    .replace(&#39;&quot;&#39;, &quot;&quot;&quot;&quot;);
    .replace(&#39;\&#39;&#39;, &quot;&#39;&quot;);
}

#[cfg(test)]
mod tests{
    usesuper::*;

    #[test]
    fn test_html_escape(){

```

a
s
s
e
r
t
-

```

eq!(html_escape(&quot;&lt;div&gt;&quot;),&quot;&amp;&lt;div&amp;gt;&quot;);
    assert_eq!(html_escape(&quot;a &amp; b&quot;),&quot;a &amp;
b&quot;);

a
s
s
e
r
t
-
e
q
!
(
h
t
m
l
-
e
s
c
a
p
e(&quot;\&quot;test\&quot;&quot;),&quot;&amp;quot;test&amp;quot;&quot;);
    }
    }

```

```

288 //! git2pdf - Convert git repositories to PDF for code review
    //!

```

```

    //! This tool clones a git repository (or uses a local path),
discovers Rust crates,
    //! classifies source files vs test files, generates
syntax-highlighted HTML,
    //! and converts them to PDF using printpdf's HTML layout
engine.

```

```

usestd::collections::BTreeMap;
usestd::fs;
usestd::path::PathBuf;

useanyhow::{Context,Result, bail};
useclap::Parser;
useprintpdf::{GeneratePdfOptions, PdfDocument, PdfSaveOptions};
usesyntect::highlighting::ThemeSet;
usesyntect::parsing::SyntaxSet;

```

```

modcrate_discovery;
modfile_classifier;
modgit_ops;
modhtml_generator;

usecrate_discovery::{CrateInfo, discover_crates};
usefile_classifier::{classify_files, SourceFile, FileCategory};
usegit_ops::{clone_or_open_repo, checkout_ref};
usehtml_generator::generate_html_for_crate;

/// git2pdf - Print git repositories to PDF for code review
#[derive(Parser, Debug)]
#[command(name = "git2pdf")]
#[command(author, version, about, long_about = None)]
struct Args {
    /// Git repository URL or local file path
    #[arg(value_name = "SOURCE")]
    source: String,

    /// Branch, tag, or commit to checkout (default: tries
    /// 'main', then 'master')
    #[arg(short, long)]
    r#ref: Option<String>,

    /// Output directory for generated PDFs (default: current
    directory)
    #[arg(short, long, default_value = ".")]
    output: PathBuf,

    /// Paper width in mm (default: 210 for A4)
    #[arg(long, default_value = "210.0")]
    paper_width: f32,

    /// Paper height in mm (default: 297 for A4)
    #[arg(long, default_value = "297.0")]
    paper_height: f32,

```

```
/// Top margin in mm
    #[arg(long, default_value = "10.0")]
margin_top:f32,

/// Right margin in mm
    #[arg(long, default_value = "10.0")]
margin_right:f32,

/// Bottom margin in mm
    #[arg(long, default_value = "10.0")]
margin_bottom:f32,

/// Left margin in mm
    #[arg(long, default_value = "10.0")]
margin_left:f32,

/// Font size in points for code
    #[arg(long, default_value = "8.0")]
font_size:f32,

/// Number of columns for code layout (default: 2)
    #[arg(long, default_value = "2")]
columns:u32,

/// Include test files in output
    #[arg(long)]
include_tests:bool,

/// Syntax highlighting theme (default: InspiredGitHub)
    #[arg(long, default_value = "InspiredGitHub")]
theme: String,

/// Verbose output
    #[arg(short, long)]
verbose:bool,
```

```

/// Only process specific crates (comma-separated)
    #[arg(long)]
crates:Option<String>;

/// Temporary directory for cloning (default: system temp)
    #[arg(long)]
temp_dir:Option<PathBuf>;
}

fnmain()->Result<()>{
    let args =Args::parse();

    if args.verbose {
        println!(&quot;git2pdf - Converting repository to PDF&quot;);
        println!(&quot;Source: {}&quot;;, args.source);
    }

    // Determine if source is a URL or local path
    let repo_path =if args.source.starts_with(&quot;http://&quot;);
    || args.source.starts_with(&quot;https://&quot;);
    || args.source.starts_with(&quot;git@&quot;);
    || args.source.starts_with(&quot;ssh://&quot;);
    {
        // Clone the repository
        let temp_dir = args.temp_dir.clone().unwrap_or_else(||{
            std::env::temp_dir().join(&quot;git2pdf&quot;);
        });
        fs::create_dir_all(&temp_dir)?;

        let repo_name =extract_repo_name(&args.source)?;
        let clone_path = temp_dir.join(&repo_name);

        if args.verbose {
            println!(&quot;Cloning to: {}&quot;;, clone_path.display());
        }
    }
}

```

```

clone_or_open_repo(&args.source,&clone_path,
args.verbose)?;
    clone_path
}else{
// Use local path
PathBuf::from(&args.source)
};

if!repo_path.exists(){
    bail!(&"Repository path does not exist: {}&",
repo_path.display());
}

// Checkout the specified ref if provided
ifletSome(ref git_ref)= args.r#ref{
    if args.verbose {
println!(&"Checking out: {}&", git_ref);
    }
    checkout_ref(&repo_path, git_ref, args.verbose)?;
}

// Discover crates in the repository
if args.verbose {
println!(&"Discovering crates...&");
}
let crates =discover_crates(&repo_path)?;

if crates.is_empty(){
    bail!(&"No Rust crates found in repository&");
}

if args.verbose {
println!(&"Found {} crate(s):&", crates.len());
for c in&crates {
println!(&" - {} ({})&", c.name, c.path.display());
}
}

```

```

}

// Filter crates if specified
let crates_to_process:Vec<&CrateInfo>=if let Some(ref
filter)= args.crates {
    let filter_names:Vec<&str>=
filter.split(&#39;&#39;).map(|s|s.trim()).collect();
    crates.iter()
        .filter(|c|filter_names.contains(&c.name.as_str()))
        .collect()
}else{
    crates.iter().collect()
};

if crates_to_process.is_empty(){
    bail!(&quot;No crates matched the filter&quot;);
}

// Create output directory
fs::create_dir_all(&args.output)?;

// Load syntax highlighting
let syntax_set =SyntaxSet::load_defaults_newlines();
let theme_set =ThemeSet::load_defaults();
let theme = theme_set.themes.get(&args.theme)

.or_else(||theme_set.themes.get(&quot;InspiredGitHub&quot;))
    .context(&quot;Failed to load syntax theme&quot;)?;

// Process each crate
for crate_info in crates_to_process {
    if args.verbose {
        println!(&quot;\nProcessing crate: {}&quot;, crate_info.name);
    }

    // Classify files
    let files =classify_files(&crate_info.path,
args.include_tests)?;

```

```

let source_files:Vec<&SourceFile>= files.iter()
    .filter(|f|f.category ==FileCategory::Source ||
        (args.include_tests
&&matches!(f.category,FileCategory::Test
|FileCategory::IntegrationTest)))
    .collect();

if source_files.is_empty(){
if args.verbose {
println!(&quot; No source files found, skipping&quot;);
}
continue;
}

if args.verbose {
println!(&quot; Found {} source file(s)&quot;,
source_files.len());
}

// Generate HTML
let html =generate_html_for_crate(
    crate_info,
    &source_files,
    &syntax_set,
    theme,
    args.font_size,
    args.columns,
)?;

// Generate PDF
let options = GeneratePdfOptions {
    page_width:Some(args.paper_width),
    page_height:Some(args.paper_height),
    margin_top:Some(args.margin_top),
    margin_right:Some(args.margin_right),
    margin_bottom:Some(args.margin_bottom),
    margin_left:Some(args.margin_left),

```

```

        show_page_numbers: Some(true),
        header_text: Some(format!("{}", " - Code Review",
crate_info.name)),
        ..Default::default()
    };

    let images = BTreeMap::new();
    let fonts = BTreeMap::new();
    letmut warnings = Vec::new();

    let doc
=
P
d
f
Document::from_html(&html, &images, &fonts, &options, &mut
warnings)
        .map_err(|e| anyhow::anyhow!("{}", "Failed to generate
PDF: {}", e))?;

    if args.verbose && !warnings.is_empty(){
        println!("{}", "PDF generation warnings: {}",
warnings.len());
    }

    // Save PDF

    let output_path = args.output.join(format!("{}", ".pdf",
crate_info.name));
    let save_options = PdfSaveOptions::default();
    letmut save_warnings = Vec::new();
    let bytes = doc.save(&save_options, &mut save_warnings);

    fs::write(&output_path, bytes)?;
    println!("{}", "Created: {}", output_path.display());
}

println!("{}", "\nDone!");
Ok(())
}

/// Extract repository name from URL

```

```

fnextract_repo_name(url:&str)->Result<String>{
    // Handle various URL formats:
    // https://github.com/user/repo.git
    // git@github.com:user/repo.git
    // ssh://git@github.com/user/repo.git

    let url = url.trim_end_matches(".git");

    ifletSome(name)= url.rsplit("#39;/#39;).next(){
        if!name.is_empty(){
            returnOk(name.to_string());
        }
    }

    // Try git@ format
    ifletSome(path)= url.split("#39;:#39;).last(){
        ifletSome(name)= path.rsplit("#39;/#39;).next(){
            if!name.is_empty(){
                returnOk(name.to_string());
            }
        }
    }

    bail!("Could not extract repository name from URL: {}",
url)
}

```