Code Repository Documentation

Repository: src

Generated: 4/30/2025, 2:29:08 PM

Table of Contents

cli.ts

```typescript
#!/usr/bin/env node

import { Command, OptionValues } from 'commander';
import path from 'path';
import fs from 'fs-extra';
import { run } from './main';
import { logger } from './utils/logger';
import { PdfOptions } from './utils/types';
import { themes } from './utils/themes'; // Import available themes for validation

/**
 * Reads the package version from package.json.
 * Handles potential errors during file reading.
 * @returns The package version string or a fallback.
 */
function getPackageVersion(): string {
    let packageVersion = '0.0.0'; // Default fallback version
    try {
        // Resolve path relative to the executing JS file (expected in dist/)
        const packageJsonPath = path.resolve(__dirname, '..', 'package.json');
        if (fs.existsSync(packageJsonPath)) {
             const packageJson = fs.readJsonSync(packageJsonPath);
             packageVersion = packageJson.version || packageVersion;
        } else {
            // This might happen during development if 'dist' doesn't exist yet
            logger.debug(`package.json not found at expected path: ${packageJsonPath}`);
        }
    } catch (error) {
        // Log warning but don't crash if package.json is unreadable
        logger.warn(`Could not read package.json: ${(error as Error).message}`);
    }
    return packageVersion;
}

/**
 * Creates and configures the Commander program for the CLI, defining arguments and options.
 * @returns The configured Commander program instance.
 */
function setupCli(): Command {
    const program = new Command();
    const packageVersion = getPackageVersion();

    program
        .name('xprinto')
        .description('Convert code repositories or directories to beautiful PDFs with syntax highlighting.')
        .version(packageVersion)
        .argument('<repository-path>', 'Path to the code repository or directory to process')
        .option('-o, --output <path>', 'Output path for the generated PDF file.', 'code-output.pdf')
        .option('-t, --title <title>', 'Title for the PDF document cover page.', 'Code Repository Documentation')
        .option('-f, --font-size <size>', 'Font size (in points) for code blocks.', '9')
        .option('--theme <name>', `Syntax highlighting theme (available: ${Object.keys(themes).join(', ')}).`, 'light')

  // Default is true, --no-line-numbers flag makes it false via Commander's boolean handling
        .option('--line-numbers', 'Show line numbers in code blocks (default).', true)
        .option('--no-line-numbers', 'Hide line numbers in code blocks.')
        .option('--paper-size <size>',
```

cli.ts

```
  'Paper size (A4, Letter, or width,height in points e.g., "595.28,841.89").', 'A4')
        .option('-v, --verbose', 'Enable verbose (debug) logging output.', false)
        .action(runCliAction); // Delegate the core logic to the action function

    return program;
}

/**
 * Validates parsed command-line options and constructs the PdfOptions object.
 * Logs errors and exits the process with code 1 if validation fails.
 * @param repoPathArg The repository path argument provided by the user.
 * @param options The parsed options object from Commander.
 * @returns A Promise resolving to an object containing the validated PdfOptions and the resolved repository path.
 */
async function validateAndPrepareOptions(repoPathArg: string, options: OptionValues): Promise<{ resolvedRepoPath: string; pdfOptions: PdfOptions }> {
    // Set logger verbosity based on the --verbose flag
    logger.setVerbose(options.verbose);

    // Resolve paths to absolute paths for consistency
    const resolvedRepoPath = path.resolve(process.cwd(), repoPathArg); // Resolve relative to current working directory
    const resolvedOutputPath = path.resolve(process.cwd(), options.output);

    logger.info(`Input path resolved to: ${resolvedRepoPath}`);
    logger.info(`Output path resolved to: ${resolvedOutputPath}`);

    // --- Validate Input Path ---
    try {
        const stats = await fs.stat(resolvedRepoPath);
        if (!stats.isDirectory()) {
            logger.error(`'L Input path must be a directory${resolvedRepoPath}`);
            process.exit(1); // Exit on validation failure
        }
    } catch (error) {
        logger.error(`'L Cannot access input path${resolvedRepoPath}`);
        if ((error as NodeJS.ErrnoException).code === 'ENOENT') {
             logger.error("   Reason: Path does not exist.");
        } else {
             logger.error(`   Reason: ${(error as Error).message}`);
        }
        process.exit(1); // Exit on validation failure
    }

    // --- Validate Theme ---
    const themeName = options.theme.toLowerCase();
    if (!themes[themeName]) {
         logger.error(`'L Invalid theme specified:$"{options.theme}".`);
         logger.error(`   Available themes: ${Object.keys(themes).join(', ')}`);
         process.exit(1); // Exit on validation failure
    }

    // --- Parse and Validate Paper Size ---
    let paperSizeOption: PdfOptions['paperSize'];
    const paperSizeInput = options.paperSize;
    if (paperSizeInput.includes(',')) {
        const dims = paperSizeInput.split(',').map(Number);
        if (dims.length === 2 && !isNaN(dims[0]) && !isNaN(dims[1]) && dims[0] > 0 && dims[1] > 0) {
```

cli.ts

```
            paperSizeOption = [dims[0], dims[1]];
            logger.debug(`Using custom paper size: ${dims[0]}x${dims[1]} points.`);
        } else {
            logger.error(`'L Invalid custom paper size format:${paperSizeInput}
  ". Use "width,height" in positive points (e.g., "595.28,841.89").`);
            process.exit(1); // Exit on validation failure
        }
    } else if (paperSizeInput.toUpperCase() === 'A4' || paperSizeInput.toUpperCase() ===
  'LETTER') {
        paperSizeOption = paperSizeInput.toUpperCase() as 'A4' | 'Letter';
        logger.debug(`Using standard paper size: ${paperSizeOption}`);
    } else {
        logger.error(`'L Invalid paper size name:${paperSizeInput}
  ". Use "A4", "Letter", or "width,height".`);
        process.exit(1); // Exit on validation failure
    }

    // --- Parse and Validate Font Size ---
    const fontSize = parseInt(options.fontSize, 10);
    // Add reasonable bounds check for font size
    if (isNaN(fontSize) || fontSize <= 2 || fontSize > 72) {
        logger.error(`'L Invalid font size:${options.fontSize}
  ". Must be a positive number (e.g., 8-14 recommended).`);
        process.exit(1); // Exit on validation failure
    }

    // --- Construct Final Options Object ---
    const pdfOptions: PdfOptions = {
        output: resolvedOutputPath,
        title: options.title,
        fontSize: fontSize,

  // Commander automatically handles boolean flags like --line-numbers / --no-line-numbers
        showLineNumbers: options.lineNumbers,
        theme: themeName,
        paperSize: paperSizeOption,
        // Define sensible defaults for layout - could be made configurable if needed
        margins: { top: 50, right: 40, bottom: 50, left: 40 },
        headerHeight: 25, // Space reserved for header (file path)
        footerHeight: 25, // Space reserved for footer (page number)
        tocTitle: "Table of Contents",
        codeFont: 'Courier', // Standard monospace PDF font (widely available)
        textFont: 'Helvetica' // Standard sans-serif PDF font (widely available)
    };

    // Return validated options and resolved path
    return { resolvedRepoPath, pdfOptions };
}


/**
 * The main action function executed by Commander when the CLI command is run.
 * It orchestrates option validation and calls the core application logic (`run`).
 * Handles top-level errors and sets the process exit code appropriately.
 * @param repoPathArg The repository path argument provided by the user.
 * @param options The parsed options object from Commander.
 */
async function runCliAction(repoPathArg: string, options: OptionValues): Promise<void> {
    try {
        // Validate inputs and prepare the options object needed by the core logic
```

cli.ts

```
        const { resolvedRepoPath, pdfOptions } = await validateAndPrepareOptions
  (repoPathArg, options);

        // Execute the main application logic from main.ts
        await run(resolvedRepoPath, pdfOptions);

        // If 'run' completes without throwing, log final success message
        logger.info("'  Process completed successfully)";

    } catch (error) {
        // Catch errors propagated from 'run' or validation steps
        // Specific error messages should have already been logged by the logger
        logger.error("'L Process failed due to an error)";
        // Ensure the node process exits with a non-zero code to indicate failure
        process.exitCode = 1;
    }
}

// --- Execute CLI ---
/**
 * Entry point check: Only run the CLI setup and parsing logic
 * if this script is the main module being executed (i.e., not imported elsewhere).
 */
if (require.main === module) {
    const cli = setupCli();
    cli.parse(process.argv); // Parse command-line arguments and execute action
} else {
    // This block usually won't run when executed as a CLI tool,
    // but useful if exporting setupCli for testing.
    logger.debug("CLI setup skipped (not main module).");
}

```

file-finder.ts

```
import path from 'path';
import fs from 'fs-extra'; // Using fs-extra for convenience like pathExists, readFile, stat
import { glob } from 'glob';
import ignore, { Ignore } from 'ignore'; // Note: 'ignore' package includes its own types
import { logger } from './utils/logger';
import { FileInfo } from './utils/types';

/**
 * Set of common binary file extensions to exclude from processing.
 * This list can be expanded based on typical project structures.
 */
const BINARY_EXTENSIONS = new Set([
    // Images
    'png', 'jpg', 'jpeg', 'gif', 'bmp', 'tiff', 'webp', 'ico',
    // Audio
    'mp3', 'wav', 'ogg', 'flac', 'aac', 'm4a',
    // Video
    'mp4', 'avi', 'mov', 'wmv', 'mkv', 'webm', 'flv',
    // Documents
    'pdf', 'doc', 'docx', 'xls', 'xlsx', 'ppt', 'pptx', 'odt', 'ods', 'odp',
    // Archives
    'zip', 'rar', 'gz', 'tar', '7z', 'bz2', 'xz', 'iso', 'dmg',
    // Executables & Libraries
    'exe', 'dll', 'so', 'dylib', 'app', 'msi', 'deb', 'rpm',
    // Compiled code / Intermediate files
    'o', 'a', 'obj', 'class', 'pyc', 'pyd', 'jar', 'war', 'ear',
    // Fonts
    'ttf', 'otf', 'woff', 'woff2', 'eot',
    // Databases
    'db', 'sqlite', 'sqlite3', 'mdb', 'accdb', 'dump', 'sqlitedb',
    // Other common non-text files
    'lock', // Lock files (e.g., package-lock.json is text, but yarn.lock might be handled differently)
    'log', // Log files (often large and not source code)
    'svg', // Often treated as code, but can be large assets; exclude for safety unless needed
    'DS_Store', // macOS metadata
    'bin', // Generic binary extension
    'dat', // Generic data extension
    // Add more as needed
]);

/**
 * Glob patterns for files/directories to always ignore, regardless of .gitignore content.
 * Uses gitignore pattern syntax. Ensures common build artifacts, dependencies, and metadata are skipped.
 */
const ALWAYS_IGNORE = [
    '**/node_modules/**',
    '**/.git/**',
    '**/.svn/**',
    '**/.hg/**',
    '**/.bzr/**',
    '**/.DS_Store',
    // Common build/output directories
    '**/dist/**',
    '**/build/**',
    '**/out/**',
    '**/target/**', // Java/Rust common target dir
    '**/.next/**', // Next.js build output
```

file-finder.ts

```
    '**/.nuxt/**', // Nuxt.js build output
    '**/.svelte-kit/**', // SvelteKit build output
    // Common dependency/cache directories
    '**/bower_components/**',
    '**/jspm_packages/**',
    '**/vendor/**', // PHP Composer, Go modules etc.
    '**/.cache/**',
    '**/.npm/**',
    '**/.yarn/**',
    // Common IDE/Editor directories
    '**/.vscode/**',
    '**/.idea/**',
    '**/*.swp', // Vim swap files
    '**/*.swo', // Vim swap files
    '**/.project', // Eclipse
    '**/.settings', // Eclipse
    '**/.classpath', // Eclipse
    // Common OS/Tooling files
    '**/Thumbs.db',
    '**/.env', // Environment variables often contain secrets
    '**/.env.*',
    // Common log/report directories
    '**/logs/**',
    '**/coverage/**',
    '**/report*/**', // Common report directories
];

/**
 * Checks if file content appears to be binary.
 * This is a heuristic based on the presence of null bytes, which are uncommon in UTF-8
->  text files.
 * @param content The file content as a string.
 * @returns True if the content likely contains binary data, false otherwise.
 */
function isLikelyBinary(content: string): boolean {
    // A simple check for the NULL character (\u0000).
    // While not foolproof, it catches many common binary file types.
    return content.includes('\u0000');
}

/**
 * Asynchronously reads and parses all relevant .gitignore files within a repository path.
 * Handles nested .gitignore files and correctly interprets paths relative to their location.
 * @param repoPath The absolute path to the repository root.
 * @param ig The `ignore` instance to add the loaded rules to.
 */
async function loadGitignoreRules(repoPath: string, ig: Ignore): Promise<void> {
    // Find all .gitignore files, excluding globally ignored directories for efficiency
    const gitignoreFiles = await glob('**/.gitignore', {
        cwd: repoPath,
        absolute: true,
        dot: true,
        ignore: ALWAYS_IGNORE,
        follow: false, // Do not follow symlinks
    });

    logger.debug(`Found ${gitignoreFiles.length} .gitignore files to process.`);

    // Process each found .gitignore file
    for (const gitignorePath of gitignoreFiles) {
```

file-finder.ts

```
    try {
      // Double-check existence in case glob found a broken link etc.
      if (await fs.pathExists(gitignorePath)) {
        const content = await fs.readFile(gitignorePath, 'utf-8');
        // Determine the directory of the .gitignore relative to the repo root
        const relativeDir = path.dirname(path.relative(repoPath, gitignorePath));

        // Parse lines, handling comments, empty lines, and path relativity
        const rules = content.split(/\r?\n/).map(line => {
          const trimmedLine = line.trim();
          // Ignore comments (#) and empty lines
          if (!trimmedLine || trimmedLine.startsWith('#')) {
            return ''; // Return empty string for filtering
          }
          // Handle paths relative to the .gitignore file's location
          // If a pattern doesn't start with '/' and the .gitignore isn't in the root, prepend its directory.
          // This matches standard gitignore behavior.
          if (!trimmedLine.startsWith('/') && relativeDir !== '.') {
            // Handle negation patterns ('!') correctly by prepending dir *after* the '!'
            if (trimmedLine.startsWith('!')) {
              // Use path.posix.join for consistent forward slashes
              return '!' + path.posix.join(relativeDir, trimmedLine.substring(1));
            } else {
              return path.posix.join(relativeDir, trimmedLine);
            }
          }
          // Use the line as is (it's absolute from repo root, or relativity handled)
          // Ensure forward slashes for consistency with 'ignore' package expectations
          return trimmedLine.replace(/\\/g, '/');
        }).filter(Boolean); // Remove empty strings from comments/blank lines

        // Add the parsed rules to the ignore instance
        if (rules.length > 0) {
          ig.add(rules);
          logger.debug(`Loaded ${rules.length} rules from: ${gitignorePath}`);
        }
      }
    } catch (error) {
      // Log errors reading/parsing specific gitignore files but continue processing others
      logger.warn(`Failed to read or parse .gitignore file ${gitignorePath}: ${(error as Error).message}`);
    }
  }
}

/**
 * Finds relevant code files within a given repository path.
 * It respects .gitignore rules, filters out binary files, skips overly large files,
 * and ignores common non-code directories/files.
 * @param repoPath The absolute path to the repository root directory.
 * @returns A promise resolving to an array of FileInfo objects for included files, sorted alphabetically.
 * @throws An error if the initial path cannot be accessed or is not a directory.
```

file-finder.ts

```typescript
 */
export async function findCodeFiles(repoPath: string): Promise<FileInfo[]> {
    logger.info(`Scanning directory: ${repoPath}`);

    // --- 1. Validate repoPath ---
    try {
        const stats = await fs.stat(repoPath);
        if (!stats.isDirectory()) {
            // Throw a specific error if the path isn't a directory
            throw new Error(`Input path is not a directory: ${repoPath}`);
        }
    } catch (error) {
        logger.error(`Error accessing input path ${repoPath}: ${(error as Error).message}`);
        // Re-throw the error to halt execution if the path is invalid
        throw error;
    }

    // --- 2. Initialize ignore instance and load rules ---
    const ig = ignore();
    ig.add(ALWAYS_IGNORE); // Add global ignores first
    await loadGitignoreRules(repoPath, ig); // Load all .gitignore rules

    // --- 3. Find all potential files using glob ---
    // Use stat:true to get file size efficiently during globbing
    const allFilePaths = await glob('**/*', {
        cwd: repoPath,
        absolute: true,
        nodir: true, // Only files
        dot: true, // Include dotfiles
        follow: false, // Don't follow symlinks
        ignore: ['**/node_modules/**', '**/.git/**'], // Basic ignore for glob performance; main filtering is below
        stat: true, // Request stats object for size check
        withFileTypes: false, // Paths are sufficient with absolute:true and nodir:true
    });

    logger.info(`Found ${allFilePaths.length} total file system entries initially.`);

    // --- 4. Filter and process files ---
    const includedFiles: FileInfo[] = [];
    const fileSizeLimit = 10 * 1024 * 1024; // 10 MB limit (configurable?)

    // Process files potentially in parallel
    await Promise.all(allFilePaths.map(async (globResult) => {
        // The result from glob with stat:true is an object with a path property
        // However, type definitions might be simpler; cast or check type if needed.
        // For simplicity, assuming it returns path strings or objects easily usable.
        // Let's assume globResult is the path string here for clarity. Adjust if types differ.
        const absolutePath = globResult as string; // Adjust based on actual glob return type with stat:true
        const relativePath = path.relative(repoPath, absolutePath).replace(/\\/g, '/'); // Ensure forward slashes

        // --- Filtering Logic ---
        // a) Skip if ignored by .gitignore or global rules
        if (ig.ignores(relativePath)) {
            logger.debug(`Ignoring (gitignore/always): ${relativePath}`);
            return;
        }
```

file-finder.ts

```

    // b) Skip binary files based on extension
    const extension = path.extname(absolutePath).substring(1).toLowerCase();
    if (BINARY_EXTENSIONS.has(extension)) {
      logger.debug(`Ignoring (binary extension): ${relativePath}`);
      return;
    }

    // c) Read file content and perform content-based checks
    try {
      // Get stats (might be redundant if glob provides reliable stats)
      const stats = await fs.stat(absolutePath);

      // d) Skip overly large files
       if (stats.size > fileSizeLimit) {
         logger.warn(`Ignoring (large file > ${fileSizeLimit / 1024 / 1024}MB): ${relativePath}`);
         return;
       }
       // e) Skip empty files
       if (stats.size === 0) {
         logger.debug(`Ignoring (empty file): ${relativePath}`);
         return;
       }

      // f) Read content and check for binary markers
      const content = await fs.readFile(absolutePath, 'utf-8');
      if (isLikelyBinary(content)) {
        logger.debug(`Ignoring (likely binary content): ${relativePath}`);
        return;
      }

      // --- Add to included list ---
      // If all checks pass, create FileInfo object
      includedFiles.push({
        absolutePath,
        relativePath,
        content,
        extension,
        language: '', // Language detection is done later
      });
    } catch (error) {
      // Catch errors during stat or readFile (permissions, non-UTF8, etc.)
      logger.warn(`Could not read or process file ${relativePath} (skipping): ${(error as Error).message}`);
    }
  })); // End Promise.all map

  // --- 5. Sort results and return ---
  // Sort alphabetically by relative path for consistent PDF output order
  includedFiles.sort((a, b) => a.relativePath.localeCompare(b.relativePath));

  logger.success(`Found ${includedFiles.length} relevant text files to include in the PDF.`);
  return includedFiles;
}


```

main.ts

```
import path from 'path';
import { findCodeFiles } from './file-finder';
import { highlightCode } from './syntax-highlighter';
import { generatePdf } from './pdf-renderer';
import { PdfOptions, HighlightedFile, FileInfo } from './utils/types';
import { getTheme } from './utils/themes';
import { logger } from './utils/logger';

/**
 * Main orchestration function for the xprinto tool.
 * Takes the repository path and PDF options, finds files, highlights them,
 * and generates the final PDF document. Handles top-level errors.
 *
 * @param repoPath Absolute path to the repository/directory to process.
 * @param options PDF generation options derived from CLI arguments.
 * @returns A Promise
   that resolves when the process is complete or rejects on critical error.
 * @throws Propagates errors from file finding or PDF generation stages if
   they are not handled internally.
 */
// *** Added 'export' keyword here ***
export async function run(repoPath: string, options: PdfOptions): Promise<void> {
    logger.info(`Starting processing for repository: ${repoPath}`);
    logger.info(`Output PDF will be saved to: ${options.output}`);
    logger.info(`Using Theme: ${options.theme}, Font Size: ${options.fontSize}
  pt, Line Numbers: ${options.showLineNumbers}`);

    try {
        // --- Step 1: Find relevant code files ---
        logger.info("Scanning for code files...");
        const filesToProcess: FileInfo[] = await findCodeFiles(repoPath);

        // If no files are found, log a warning and exit gracefully.
        if (filesToProcess.length === 0) {
            logger.warn(
  "No relevant code files found in the specified path. Nothing to generate.");
            return; // Exit the function successfully, nothing more to do.
        }
        logger.info(`Found ${filesToProcess.length} files to process.`);

        // --- Step 2: Load the selected syntax theme ---
        const theme = getTheme(options.theme);
        logger.info(`Using theme: ${options.theme}`); // Log the name provided by the user

        // --- Step 3: Apply syntax highlighting ---
        logger.info("Applying syntax highlighting to files...");
        const highlightStartTime = Date.now();

        // Process highlighting for each file, handling individual file errors
        const highlightedFiles: HighlightedFile[] = filesToProcess.map(fileInfo => {
            try {
                 // Attempt to highlight the code for the current file
                 return highlightCode(fileInfo, theme);
            } catch (highlightError) {
                // Catch and log errors during highlighting of a single file
                logger.error(`Failed to highlight ${fileInfo.relativePath}:
  ${(highlightError as Error).message}`);

  // Return a fallback structure for this file to prevent crashing PDF generation
                // The content will appear unhighlighted in the PDF.
```

main.ts

```
                return {
                    ...fileInfo,
                    language: 'plaintext', // Mark as plaintext due to error
                    highlightedLines: fileInfo.content.split(/\r?\n/).map((line, index) => ({
                        lineNumber: index + 1,
                        tokens: [{ text: line, color: theme.defaultColor, fontStyle: 'normal' }],
                    })),
                };
            }
        });
        const highlightEndTime = Date.now();
        logger.info(`Syntax highlighting complete (${((highlightEndTime - highlightStartTime) / 1000).toFixed(2)}s).`);


        // --- Step 4: Generate the PDF document ---
        logger.info("Generating PDF document...");
        const repoName = path.basename(repoPath); // Use directory name for cover page context

        // generatePdf handles its own success/error logging for the final PDF generation step
        await generatePdf(highlightedFiles, options, theme, repoName);

    } catch (error) {
        // Catch critical errors (e.g., from file finding, PDF stream setup)
        logger.error(`'L An unexpected critical error occurred during the process`);
        logger.error((error as Error).message);
        // Log the stack trace if verbose mode is enabled for detailed debugging
        if (logger.isVerbose()) {
             console.error("Stack Trace:");
             console.error((error as Error).stack);
        }
        // Re-throw the error so the calling context (CLI) knows about the failure
        // and can set the appropriate exit code.
        throw error;
    }
}

```

pdf-renderer.ts

```
import PDFDocument from 'pdfkit';
import fs from 'fs-extra';
import path from 'path';
import { HighlightedFile, HighlightedLine, HighlightedToken, PdfOptions, SyntaxTheme } from
  './utils/types';
import { logger } from './utils/logger';

// --- Constants ---
const POINTS_PER_INCH = 72;
/** Multiplier for calculating line height based on font size for code blocks. */
const DEFAULT_LINE_HEIGHT_MULTIPLIER = 1.4;
/** Indentation (in points) for file names under directory names in the Table of Contents. */
const TOC_INDENT = 20;
/** Number of spaces used for indenting wrapped lines of code. */
const WRAP_INDENT_MULTIPLIER = 2;
/** Padding (in points) around the dot leader in the Table of Contents. */
const TOC_DOT_PADDING = 5;
/** Padding (in points) inside the code block container (around text, line numbers). */
const CODE_BLOCK_PADDING = 10;
/** Character(s) used to indicate a wrapped line in the line number gutter. */
const WRAP_INDICATOR = '->'; // Using simple ASCII

// --- Helper Functions ---

/**
 * Converts standard paper size names ('A4', 'Letter') or a [width, height] array
 * into PDF point dimensions [width, height]. Validates input and defaults to A4 on error.
 * @param size The paper size specified in PdfOptions.
 * @returns A tuple [width, height] in PDF points.
 */
function getPaperSizeInPoints(size: PdfOptions['paperSize']): [number, number] {
    if (Array.isArray(size)) {
        // Validate custom size array
        if (size.length === 2 && typeof size[0] === 'number' && typeof size[1] === 'number'
   && size[0] > 0 && size[1] > 0) {
            return size;
        } else {
            logger.warn(`Invalid custom paper size array: [${size.join(', ')}
  ]. Falling back to A4.`);
            return [595.28, 841.89]; // Default to A4
        }
    }
    // Handle standard size names
    switch (size?.toUpperCase()) { // Add safe navigation for size
        case 'LETTER':
            return [8.5 * POINTS_PER_INCH, 11 * POINTS_PER_INCH];
        case 'A4':
            return [595.28, 841.89]; // A4 dimensions in points
        default:
            // Log warning and default to A4 if string is unrecognized or null/undefined
            logger.warn(`Unrecognized paper size string: "${size}". Falling back to A4.`);
            return [595.28, 841.89];
    }
}

/**
 * Calculates the available vertical space (in points) for content on a page,
 * excluding margins, header, and footer. Ensures result is non-negative.
 * @param doc The active PDFDocument instance.
 * @param options The PDF generation options.
```

pdf-renderer.ts

```
 * @returns The calculated content height in points.
 */
function getContentHeight(doc: PDFKit.PDFDocument, options: PdfOptions): number {
    const pageHeight = doc.page.height; // Use current page height
    const calculatedHeight = pageHeight - options.margins.top - options.margins.bottom
   - options.headerHeight - options.footerHeight;
    return Math.max(0, calculatedHeight); // Ensure non-negative height
}

/**
 * Calculates the available horizontal space (in points) for content on a page,
 * excluding left and right margins. Ensures result is non-negative.
 * @param doc The active PDFDocument instance.
 * @param options The PDF generation options.
 * @returns The calculated content width in points.
 */
 function getContentWidth(doc: PDFKit.PDFDocument, options: PdfOptions): number {
    const pageWidth = doc.page.width; // Use current page width
    const calculatedWidth = pageWidth - options.margins.left - options.margins.right;
    return Math.max(0, calculatedWidth); // Ensure non-negative width
}


// --- PDF Rendering Sections ---

/**
 * Adds a cover page to the PDF document. Includes basic error handling.
 * @param doc The active PDFDocument instance.
 * @param options The PDF generation options.
 * @param repoName The name of the repository being processed, displayed on the cover.
 */
function addCoverPage(doc: PDFKit.PDFDocument, options: PdfOptions, repoName: string): void {
    try {
        doc.addPage({ margins: options.margins }); // Add page with specified margins
        const contentWidth = getContentWidth(doc, options);
        const pageHeight = doc.page.height;
        const topMargin = doc.page.margins.top;
        const bottomMargin = doc.page.margins.bottom;
        const availableHeight = pageHeight - topMargin - bottomMargin;

        // Position elements vertically relative to available height
        const titleY = topMargin + availableHeight * 0.2;
        const repoY = titleY + 50; // Adjust spacing as needed
        const dateY = repoY + 30;

        // Title
        doc.font(options.textFont + '-Bold')
           .fontSize(24)
           .text(options.title, doc.page.margins.left, titleY, {
                align: 'center',
                width: contentWidth
           });

        // Repository Name
        doc.font(options.textFont)
           .fontSize(16)
           .text(`Repository: ${repoName}`, doc.page.margins.left, repoY, {
                align: 'center',
                width: contentWidth
            });
```

pdf-renderer.ts

```

        // Generation Date
        doc.font(options.textFont) // Reset font style
           .fontSize(12)
           .fillColor('#555555') // Use a less prominent color
           .text(`Generated: ${new Date().toLocaleString()}`, doc.page.margins.left, dateY, {
               align: 'center',
               width: contentWidth
            });
        // *** REMOVED problematic line: doc.fillColor(theme.defaultColor || '#000000'); ***

        logger.info('Added cover page.');
    } catch (error) {
        logger.error(`Failed to add cover page: ${(error as Error).message}`);
        // Decide if this error should halt the process or just be logged
    }
}

/**
 * Adds a Table of Contents (TOC) page(s) to the PDF document.
 * Groups files by directory, estimates page numbers, and renders the list with dot leaders.
 * Handles page breaks within the TOC itself.
 * @param doc The active PDFDocument instance.
 * @param files An array of `HighlightedFile` objects to include in the TOC.
 * @param options The PDF generation options.
 * @param theme The active syntax theme (used for text colors).
 * @param pageNumberOffset The logical page number where the first actual code file will start.
 * @returns A record mapping file relative paths to their estimated starting page number.
 */
function addTableOfContents(
    doc: PDFKit.PDFDocument,
    files: HighlightedFile[],
    options: PdfOptions,
    theme: SyntaxTheme,
    pageNumberOffset: number
): Record<string, number> {
    const pageEstimates: Record<string, number> = {}; // Stores relativePath -> estimated startPage

    try {
        doc.addPage(); // Add the first page for the TOC
        const contentWidth = getContentWidth(doc, options);
        const startY = doc.page.margins.top;
        doc.y = startY; // Set starting Y position

        // --- TOC Title ---
        doc.font(options.textFont + '-Bold')
           .fontSize(18)
           .fillColor(theme.defaultColor)
           .text(options.tocTitle, { align: 'center', width: contentWidth });
        doc.moveDown(2); // Space after title

        // --- Group Files by Directory ---
        const filesByDir: Record<string, HighlightedFile[]> = {};
        files.forEach(file => {
            const dir = path.dirname(file.relativePath);
            const dirKey = (dir === '.' || dir === '/') ? '/' : `/${dir.replace(/\\/g, '/')}`; // Normalize key
            if (!filesByDir[dirKey]) filesByDir[dirKey] = [];
```

pdf-renderer.ts

```
            filesByDir[dirKey].push(file);
        });

        // --- Estimate Page Numbers ---
        let estimatedCurrentPage = pageNumberOffset;
        const codeLinesPerPage = Math.max(1, Math.floor(getContentHeight(doc, options) / (options.fontSize * DEFAULT_LINE_HEIGHT_MULTIPLIER)));

        const sortedDirs = Object.keys(filesByDir).sort(); // Sort directory keys alphabetically
        for (const dir of sortedDirs) {
            // Sort files within each directory alphabetically
            const sortedFiles = filesByDir[dir].sort((a, b) => a.relativePath.localeCompare(b.relativePath));
            for (const file of sortedFiles) {
                pageEstimates[file.relativePath] = estimatedCurrentPage; // Store estimated start page
                const lineCount = file.highlightedLines.length;
                // Estimate pages needed for this file (minimum 1 page)
                const estimatedPagesForFile = Math.max(1, Math.ceil(lineCount / codeLinesPerPage));
                estimatedCurrentPage += estimatedPagesForFile; // Increment estimated page counter
            }
        }
        logger.debug(`Estimated total pages after code content: ${estimatedCurrentPage - 1}`);

        // --- Render TOC Entries ---
        doc.font(options.textFont).fontSize(12); // Set default font for TOC entries
        const tocLineHeight = doc.currentLineHeight() * 1.1; // Approximate line height for TOC entries
        const tocEndY = doc.page.height - doc.page.margins.bottom; // Bottom boundary for TOC content

        for (const dir of sortedDirs) {
            // Check for page break before rendering directory header (need space for header + one entry)
            if (doc.y > tocEndY - (tocLineHeight * 2)) {
                 doc.addPage();
                 doc.y = doc.page.margins.top; // Reset Y to top margin
            }

            // Render Directory Header (if not root)
            if (dir !== '/') {
                doc.moveDown(1); // Add space before directory header
                doc.font(options.textFont + '-Bold')
                   .fillColor(theme.defaultColor)
                   .text(dir, { continued: false }); // Render directory name
                doc.moveDown(0.5); // Space after directory header
            }

            // Render File Entries for this Directory
            const sortedFiles = filesByDir[dir].sort((a, b) => a.relativePath.localeCompare(b.relativePath));
            for (const file of sortedFiles) {
                 // Check for page break before rendering file entry
                 if (doc.y > tocEndY - tocLineHeight) {
                     doc.addPage();
```

pdf-renderer.ts

```
                     doc.y = doc.page.margins.top; // Reset Y to top margin
                 }

                const fileName = path.basename(file.relativePath);
                const pageNum = pageEstimates[file.relativePath]?.toString() || '?';
  // Use estimated page
                const indent = (dir === '/') ? 0 : TOC_INDENT;
  // Indent if not in root directory
                const startX = doc.page.margins.left + indent;
                const availableWidth = contentWidth - indent;
                const currentY = doc.y;
  // Store Y position for precise placement on this line

                // Calculate positions for filename, dots, and page number
                doc.font(options.textFont).fontSize(12).fillColor(theme.defaultColor);
  // Ensure correct font for width calc
                const nameWidth = doc.widthOfString(fileName);
                const pageNumWidth = doc.widthOfString(pageNum);
                const fileNameEndX = startX + nameWidth;
                const pageNumStartX = doc.page.margins.left + contentWidth - pageNumWidth;
  // Position for right alignment

                // Render file name (ensure it doesn't wrap)
                doc.text(fileName, startX, currentY, {
                    width: nameWidth,
                    lineBreak: false,
                    continued: false // Stop after filename
                });

                // Render page number (explicitly positioned)
                doc.text(pageNum, pageNumStartX, currentY, {
                    width: pageNumWidth,
                    lineBreak: false,
                    continued: false // Stop after page number
                });

                // Render dot leader in the space between filename and page number
                const dotsStartX = fileNameEndX + TOC_DOT_PADDING;
                const dotsEndX = pageNumStartX - TOC_DOT_PADDING;
                const dotsAvailableWidth = dotsEndX - dotsStartX;

                if (dotsAvailableWidth > doc.widthOfString('. ')) {
  // Check if there's enough space for at least one dot sequence
                    const dot = '. ';
                    const dotWidth = doc.widthOfString(dot);
                    const numDots = Math.floor(dotsAvailableWidth / dotWidth);
                    const dotsString = dot.repeat(numDots);

                    doc.fillColor('#aaaaaa'); // Use a lighter color for dots
                    doc.text(dotsString, dotsStartX, currentY, {
                        width: dotsAvailableWidth, // Constrain dots width
                        lineBreak: false,
                        continued: false
                    });
                    doc.fillColor(theme.defaultColor); // Reset fill color
                }

                // Move down for the next TOC entry
                doc.moveDown(0.6); // Adjust spacing as needed
            } // End loop through files in directory
```

pdf-renderer.ts

```
        } // End loop through directories

        logger.info('Added Table of Contents.');

    } catch (error) {
        logger.error(`Failed to add Table of Contents: ${(error as Error).message}`);
        // Continue PDF generation even if TOC fails?
    }

    return pageEstimates; // Return estimates (might be useful for debugging)
}

/**
 * Renders the header section for a code page. Includes basic error handling.
 * @param doc The active PDFDocument instance.
 * @param file The `HighlightedFile` being rendered.
 * @param options The PDF generation options.
 * @param theme The active syntax theme.
 */
function renderHeader(doc: PDFKit.PDFDocument, file: HighlightedFile, options: PdfOptions,
  theme: SyntaxTheme): void {
    try {
        const headerY = doc.page.margins.top; // Use actual top margin of the current page

  // Calculate Y position to vertically center typical 9pt text within the header height
        const headerContentY = headerY + (options.headerHeight - 9) / 2;
  // Adjust multiplier if needed
        const contentWidth = getContentWidth(doc, options);
        const startX = doc.page.margins.left;

        // Draw header background rectangle
        doc.rect(startX, headerY, contentWidth, options.headerHeight)
           .fillColor(theme.headerFooterBackground)
           .fill();

        // Draw file path (truncated with ellipsis if it exceeds available width)
        doc.font(options.textFont) // Use standard text font
           .fontSize(9) // Use a smaller font size for header/footer
           .fillColor(theme.headerFooterColor)
           .text(file.relativePath, startX + CODE_BLOCK_PADDING, headerContentY, {
               width: contentWidth - (CODE_BLOCK_PADDING * 2), // Constrain width by padding
               align: 'left',
               lineBreak: false, // Prevent wrapping
               ellipsis: true // Add '...' if path is too long
           });

        // Draw border line below the header area
        doc.moveTo(startX, headerY + options.headerHeight)
           .lineTo(startX + contentWidth, headerY + options.headerHeight)
           .lineWidth(0.5) // Use a thin line
           .strokeColor(theme.borderColor)
           .stroke();
        // Reset fill color after potential changes
        doc.fillColor(theme.defaultColor || '#000000');
    } catch (error) {
         logger.error(`Failed to render header for ${file.relativePath}: ${(error as Error
  ).message}`);
    }
}

```

pdf-renderer.ts

```
/**
 * Renders the footer section for a code page. Includes basic error handling.
 * @param doc The active PDFDocument instance.
 * @param currentPage The logical page number to display.
 * @param options The PDF generation options.
 * @param theme The active syntax theme.
 */
function renderFooter(doc: PDFKit.PDFDocument, currentPage: number, options: PdfOptions, theme: SyntaxTheme): void {
     try {
        // Calculate Y position for the top of the footer area
        const footerY = doc.page.height - doc.page.margins.bottom - options.footerHeight; // Use actual bottom margin
        // Calculate Y position to vertically center typical 9pt text
        const footerContentY = footerY + (options.footerHeight - 9) / 2;
        const contentWidth = getContentWidth(doc, options);
        const startX = doc.page.margins.left;

         // Draw border line above the footer area
         doc.moveTo(startX, footerY)
            .lineTo(startX + contentWidth, footerY)
            .lineWidth(0.5)
            .strokeColor(theme.borderColor)
            .stroke();

        // Draw page number centered in the footer
        doc.font(options.textFont)
           .fontSize(9) // Use smaller font size
           .fillColor(theme.headerFooterColor)
           .text(`Page ${currentPage}`, startX, footerContentY, {
               width: contentWidth,
               align: 'center'
           });
         // Reset fill color
         doc.fillColor(theme.defaultColor || '#000000');
    } catch (error) {
         logger.error(`Failed to render footer on page ${currentPage}: ${(error as Error).message}`);
    }
}

/**
 * Renders the highlighted code content for a single file onto the PDF document.
 * Handles page breaks, line numbers (if enabled), code wrapping, and applies theme styling.
 * Manages vertical positioning explicitly to avoid overlaps.
 *
 * @param doc The active PDFDocument instance.
 * @param file The `HighlightedFile` object containing the code and tokens.
 * @param options The PDF generation options.
 * @param theme The active syntax theme.
 * @param initialPageNumber The logical page number this file should start on (used for footer).
 * @returns The last logical page number used by this file.
 */
function renderCodeFile(
    doc: PDFKit.PDFDocument,
    file: HighlightedFile,
    options: PdfOptions,
    theme: SyntaxTheme,
    initialPageNumber: number
```

pdf-renderer.ts

```
): number {

    let currentPage = initialPageNumber; // Tracks the logical page number for the footer
    const contentWidth = getContentWidth(doc, options);
    const contentHeight = getContentHeight(doc, options);
    const startY = options.margins.top + options.headerHeight; // Top of code content area
    const endY = doc.page.height - options.margins.bottom - options.footerHeight; // Bottom of code content area
    const startX = options.margins.left;
    const lineHeight = options.fontSize * DEFAULT_LINE_HEIGHT_MULTIPLIER; // Calculated line height

    // --- Calculate dimensions related to line numbers ---
    const maxLineNumDigits = String(file.highlightedLines.length).length;
    const lineNumberWidth = options.showLineNumbers
        ? Math.max(maxLineNumDigits * options.fontSize * 0.65 + CODE_BLOCK_PADDING, 35 + CODE_BLOCK_PADDING) // Ensure min width
        : 0; // No width if line numbers are disabled
    const lineNumberPaddingRight = 10; // Space between line number and start of code
    // Calculate starting X coordinate for the code text
    const codeStartX = startX + (options.showLineNumbers ? lineNumberWidth + lineNumberPaddingRight : CODE_BLOCK_PADDING);
    // Calculate the usable width for the code text (accounts for left/right padding)
    const codeWidth = contentWidth - (codeStartX - startX) - CODE_BLOCK_PADDING;
    // Indentation string and its width for wrapped lines
    const wrapIndent = ' '.repeat(WRAP_INDENT_MULTIPLIER);
    const wrapIndentWidth = doc.font(options.codeFont).fontSize(options.fontSize).widthOfString(wrapIndent);


    // --- Page Setup Helper ---

    /** Sets up the header, footer, and background visuals for a new code page. Returns the starting Y coordinate for content. */
    const setupPageVisuals = (): number => {
        try {
            renderHeader(doc, file, options, theme);
            renderFooter(doc, currentPage, options, theme); // Use the current logical page number
            const pageContentStartY = startY;
            doc.y = pageContentStartY; // Reset internal Y cursor (though we manage drawing Y manually)

            // Draw background container for the code block
            doc.rect(startX, pageContentStartY, contentWidth, contentHeight)
               .fillColor(theme.backgroundColor)
               .lineWidth(0.75)
               .strokeColor(theme.borderColor)
               .fillAndStroke(); // Fill and draw border

            // Draw line number gutter background and separator line if enabled
            if (options.showLineNumbers && lineNumberWidth > 0) {
                doc.rect(startX, pageContentStartY, lineNumberWidth, contentHeight)
                   .fillColor(theme.lineNumberBackground)
                   .fill(); // Fill gutter background
                // Draw vertical separator line
                doc.moveTo(startX + lineNumberWidth, pageContentStartY)
                   .lineTo(startX + lineNumberWidth, pageContentStartY + contentHeight)
                   .lineWidth(0.5)
                   .strokeColor(theme.borderColor)
```

pdf-renderer.ts

```
                 .stroke();
          }

           // Return the Y position where actual text content should start (includes top padding)
           return pageContentStartY + CODE_BLOCK_PADDING / 2;
      } catch (setupError) {
          logger.error(`Error setting up page visuals for ${file.relativePath}: ${(setupError as Error).message}`);
          // Return startY as a fallback, though rendering might be broken
          return startY;
      }
  };

  // --- Initial Page Setup ---
  doc.addPage(); // Add the first page for this file
  let currentLineY = setupPageVisuals(); // Set up visuals and get starting Y


  // --- Main Rendering Loop (Iterate through source lines) ---
  for (const line of file.highlightedLines) {
      const lineStartY = currentLineY; // Store the Y position where this source line begins rendering

      // --- Page Break Check ---
      // Check if rendering this line (at minimum height) would exceed the available content area
      if (lineStartY + lineHeight > endY - CODE_BLOCK_PADDING) {
           doc.addPage(); // Add a new page
           currentPage++; // Increment the logical page number
           currentLineY = setupPageVisuals(); // Set up visuals and get new starting Y
      }

      // --- Draw Line Number ---
      if (options.showLineNumbers && lineNumberWidth > 0) {
          try {
              // Determine a visible color for the line number, fallback to gray
              const lnColor = (theme.lineNumberColor && theme.lineNumberColor !== theme.lineNumberBackground)
                              ? theme.lineNumberColor
                              : '#888888';
              const numStr = String(line.lineNumber).padStart(maxLineNumDigits, ' '); // Format number string
              const numX = startX + CODE_BLOCK_PADDING / 2; // X position within padding
              const numWidth = lineNumberWidth - CODE_BLOCK_PADDING; // Available width in gutter

              doc.font(options.codeFont) // Ensure correct font
                 .fontSize(options.fontSize)
                 .fillColor(lnColor)
                 .text(numStr, numX, currentLineY, { // Draw at current line's Y
                     width: numWidth,
                     align: 'right', // Align number to the right of the gutter
                     lineBreak: false // Prevent number from wrapping
                 });
          } catch (lnError) {
               logger.warn(`Error drawing line number ${line.lineNumber} for ${file.relativePath}: ${(lnError as Error).message}`);
          }
      }

```

pdf-renderer.ts

```
        // --- Render Code Tokens (Handles Wrapping Internally) ---
        let currentX = codeStartX; // Reset X position for the start of code content for this line
        let isFirstTokenOfLine = true; // Reset wrap flag for each new source line

        /** Helper function to advance Y position and handle page breaks during line wrapping. */
        const moveToNextWrapLine = () => {
            currentLineY += lineHeight; // Advance our managed Y position
            // Check if the *new* position requires a page break
            if (currentLineY + lineHeight > endY - CODE_BLOCK_PADDING) {
                doc.addPage();
                currentPage++;
                currentLineY = setupPageVisuals(); // Setup new page, get new starting Y
            }
            // Set X for the wrapped line, applying indentation
            currentX = codeStartX + wrapIndentWidth;
            // Draw wrap indicator in the line number gutter
            if (options.showLineNumbers && lineNumberWidth > 0) {
                try {
                    const wrapColor = (theme.lineNumberColor && theme.lineNumberColor !== theme.lineNumberBackground)
                                    ? theme.lineNumberColor
                                    : '#888888';
                    doc.font(options.codeFont).fontSize(options.fontSize).fillColor(wrapColor)
                       .text(WRAP_INDICATOR, startX + CODE_BLOCK_PADDING / 2, currentLineY, { // Draw at the new Y
                           width: lineNumberWidth - CODE_BLOCK_PADDING,
                           align: 'right',
                           lineBreak: false
                       });
                } catch (wrapIndicatorError) {
                     logger.warn(`Error drawing wrap indicator for ${file.relativePath}: ${(wrapIndicatorError as Error).message}`);
                }
            }
        };

        // --- Token Loop (Iterate through tokens of the current source line) ---
        for (const token of line.tokens) {
            try {
                // Set font and color for the current token
                doc.font(options.codeFont + (token.fontStyle === 'bold' ? '-Bold' : token.fontStyle === 'italic' ? '-Oblique' : ''))
                   .fontSize(options.fontSize)
                   .fillColor(token.color || theme.defaultColor);

                const tokenText = token.text;
                // Skip empty tokens
                if (!tokenText || tokenText.length === 0) {
                    continue;
                }
                const tokenWidth = doc.widthOfString(tokenText);

                // --- Wrapping Logic ---
                if (currentX + tokenWidth <= codeStartX + codeWidth) {
                    // Token fits: Draw it and advance X
                    doc.text(tokenText, currentX, currentLineY, { continued: true, lineBreak: false });
```

pdf-renderer.ts

```
                    currentX += tokenWidth;
                } else {
                    // Token needs wrapping: Process it segment by segment
                    let remainingText = tokenText;

                    // Move to the next line in the PDF before drawing the wrapped part,
                    // but only if necessary (first token overflow or subsequent tokens).
                    if (isFirstTokenOfLine && currentX === codeStartX) {
                         moveToNextWrapLine(); // First token overflows immediately
                    } else if (!isFirstTokenOfLine) {
                         // Subsequent token overflows
                         moveToNextWrapLine();
                    }
                    // If first token partially fit, loop handles moves.

                    // Loop to draw segments of the remaining text
                    while (remainingText.length > 0) {
                        let fitsChars = 0;
                        let currentSegmentWidth = 0;
                        const availableWidth = (codeStartX + codeWidth) - currentX; // Width available

                        // Determine how many characters fit
                        for (let i = 1; i <= remainingText.length; i++) {
                            const segment = remainingText.substring(0, i);
                            const width = doc.widthOfString(segment);
                            if (width <= availableWidth + 0.001) { // Tolerance
                                fitsChars = i;
                                currentSegmentWidth = width;
                            } else {
                                break;
                            }
                        }

                        // Handle cases where not even one character fits
                        if (fitsChars === 0 && remainingText.length > 0) {
                            if (availableWidth <= 0) {
                                // No space left, definitely move to next line and retry fitting
                                moveToNextWrapLine();
                                continue; // Re-evaluate fitting in the next iteration
                            } else {
                                // Force at least one character if space was available
                                fitsChars = 1;
                                currentSegmentWidth = doc.widthOfString(remainingText[0]);
                                logger.warn(`Forcing character fit '${remainingText[0]}' on wrapped line ${line.lineNumber} of ${file.relativePath}.`);
                            }
                        }

                        // Draw the segment that fits
                        const textToDraw = remainingText.substring(0, fitsChars);
                        doc.font(options.codeFont + (token.fontStyle === 'bold' ? '-Bold' : token.fontStyle === 'italic' ? '-Oblique' : ''))
                           .fontSize(options.fontSize)
                           .fillColor(token.color || theme.defaultColor);
                        doc.text(textToDraw, currentX, currentLineY, { continued: true, lineBreak: false });

                        // Update state for the next segment/token
```

pdf-renderer.ts

```
                        currentX += currentSegmentWidth;
                        remainingText = remainingText.substring(fitsChars);


                        // If there's still remaining text in this token, move to the next line
                        if (remainingText.length > 0) {
                            moveToNextWrapLine();
                        }
                    } // End while(remainingText)
                } // End else (wrapping needed)
            } catch (tokenError) {
                 logger.warn(`Error rendering token "${token.text.substring(0, 20)}..." on line ${line.lineNumber} of ${file.relativePath}: ${(tokenError as Error).message}`);
                 // Continue to next token
            } finally {
                 isFirstTokenOfLine = false; // Mark that we are past the first token for this source line
            }
        } // End for loop (tokens)

        // --- Advance Y for Next Source Line ---

        // After processing all tokens for the original source line, move our managed Y position down.
        currentLineY += lineHeight;

    } // End for loop (lines)

    logger.info(`Rendered file ${file.relativePath} spanning pages ${initialPageNumber}-${currentPage}.`);
    return currentPage; // Return the last logical page number used by this file
}


// --- Main PDF Generation Function ---

/**
 * Orchestrates the entire PDF generation process:
 * Finds files, highlights code, sets up the PDF document, adds cover page,
 * adds table of contents (if applicable), renders each file's code, and saves the PDF.
 * Includes error handling for stream operations.
 *
 * @param files An array of `HighlightedFile` objects already processed by the syntax highlighter.
 * @param options The `PdfOptions` controlling the generation process.
 * @param theme The active `SyntaxTheme` object.
 * @param repoName The name of the repository, used for the cover page.
 * @returns A Promise that resolves when the PDF has been successfully written, or rejects on error.
 * @throws Propagates errors from critical stages like stream writing or PDF finalization.
 */
export async function generatePdf(
    files: HighlightedFile[],
    options: PdfOptions,
    theme: SyntaxTheme,
    repoName: string
): Promise<void> {
    logger.info(`Starting PDF generation for ${files.length} files.`);
    const startTime = Date.now();
```

pdf-renderer.ts

```typescript

    let doc: PDFKit.PDFDocument | null = null;
    let writeStream: fs.WriteStream | null = null;

    // Promise wrapper to handle stream events correctly
    return new Promise(async (resolve, reject) => {
        try {
            // Initialize PDF document
            doc = new PDFDocument({
                size: getPaperSizeInPoints(options.paperSize),
                margins: options.margins,
                autoFirstPage: false,
                bufferPages: true, // Enable buffering for potential page counting/manipulation
                info: { // PDF metadata
                    Title: options.title,
                    Author: 'xprinto', // Consider making this configurable
                    Creator: 'xprinto',
                    CreationDate: new Date(),
                }
            });

            // Setup file stream and pipe PDF output to it
            const outputDir = path.dirname(options.output);
            await fs.ensureDir(outputDir); // Ensure output directory exists
            writeStream = fs.createWriteStream(options.output);
            doc.pipe(writeStream);

            // --- Register Stream Event Handlers ---
            // Handle successful completion
            writeStream.on('finish', () => {
                const endTime = Date.now();
                logger.success(`PDF generated successfully: ${options.output}`);
                logger.info(`Total generation time: ${((endTime - startTime) / 1000).toFixed(2)} seconds.`);
                resolve(); // Resolve the main promise on successful finish
            });

            // Handle errors during writing
            writeStream.on('error', (err) => {
                logger.error(`WriteStream error for ${options.output}: ${err.message}`);
                reject(err); // Reject the main promise on stream error
            });

            // Handle potential errors from the PDFDocument itself
            doc.on('error', (err) => {
                logger.error(`PDFDocument error: ${err.message}`);
                reject(err); // Reject the main promise on document error
            });

            // --- Add PDF Content ---
            let physicalPageCount = 0; // Track actual pages added to the document

            // 1. Cover Page
            addCoverPage(doc, options, repoName);
            physicalPageCount = doc.bufferedPageRange().count;

            // 2. Table of Contents
            let tocPages = 0;
            let fileStartLogicalPageNumber = physicalPageCount + 1;
```

pdf-renderer.ts

```
// Logical page files start on

            if (files.length > 1) {
                const tocStartPhysicalPage = physicalPageCount + 1;
                addTableOfContents(doc, files, options, theme, fileStartLogicalPageNumber);
                const tocEndPhysicalPage = doc.bufferedPageRange().count;
                tocPages = tocEndPhysicalPage - physicalPageCount;
                physicalPageCount = tocEndPhysicalPage;
                fileStartLogicalPageNumber = physicalPageCount + 1; // Update logical start page after TOC
                logger.info(`Table of Contents added (${tocPages} page(s)). Files will start on logical page ${fileStartLogicalPageNumber}. Current physical page count: ${physicalPageCount}`);
            } else {
                 logger.info('Skipping Table of Contents (single file).');
            }

            // 3. Render Code Files
            let lastLogicalPageNumber = physicalPageCount; // Initialize with page count after cover/TOC

            const sortedFiles = files.sort((a, b) => a.relativePath.localeCompare(b.relativePath));

            for (const file of sortedFiles) {
                const currentFileStartLogicalPage = lastLogicalPageNumber + 1;
                logger.debug(`Rendering file: ${file.relativePath}, starting on logical page ${currentFileStartLogicalPage}`);
                // renderCodeFile handles adding pages internally and returns the last logical page number used
                lastLogicalPageNumber = renderCodeFile(doc, file, options, theme, currentFileStartLogicalPage);
            }

            // --- Finalize PDF ---
            logger.info("Finalizing PDF document...");
            // This triggers the 'finish' event on the writeStream eventually
            doc.end();

        } catch (error) {
             // Catch synchronous errors during setup or file processing loops
             logger.error(`Failed during PDF generation setup or rendering loop: ${(error as Error).message}`);
             // Ensure stream is closed if open, though pdfkit might handle this on error
             if (writeStream && !writeStream.closed) {
                 writeStream.close();
             }
             reject(error); // Reject the main promise
        }
    }); // End Promise wrapper
}
```

syntax-highlighter.ts

```typescript
import hljs from 'highlight.js';
import he from 'he'; // Use 'he' library for robust HTML entity decoding
import { FileInfo, HighlightedFile, HighlightedLine, HighlightedToken, SyntaxTheme } from
  './utils/types';
import { logger } from './utils/logger';

// --- Language Mapping ---

/**
 * A mapping from common file extensions (lowercase) to the language identifier
 * expected by highlight.js. This helps when highlight.js might not automatically
 * detect the correct language based solely on the extension.
 */
const LANGUAGE_MAP: Record<string, string> = {
    'ts': 'typescript',
    'tsx': 'typescript',
    'js': 'javascript',
    'jsx': 'javascript',
    'mjs': 'javascript',
    'cjs': 'javascript',
    'py': 'python',
    'pyw': 'python',
    'rb': 'ruby',
    'java': 'java',
    'cs': 'csharp',
    'go': 'go',
    'php': 'php',
    'html': 'html',
    'htm': 'html',
    'css': 'css',
    'scss': 'scss',
    'sass': 'scss', // Treat sass as scss for highlighting
    'less': 'less',
    'json': 'json',
    'yaml': 'yaml',
    'yml': 'yaml',
    'md': 'markdown',
    'sh': 'bash',
    'bash': 'bash',
    'zsh': 'bash',
    'ksh': 'bash',
    'fish': 'bash', // Highlight most shells as bash
    'sql': 'sql',
    'xml': 'xml',
    'kt': 'kotlin',
    'kts': 'kotlin',
    'swift': 'swift',
    'pl': 'perl',
    'pm': 'perl',
    'rs': 'rust',
    'lua': 'lua',
    'dockerfile': 'dockerfile',
    'h': 'c', // Often C or C++ header, default to C
    'hpp': 'cpp',
    'cpp': 'cpp',
    'cxx': 'cpp',
    'cc': 'cpp',
    'c': 'c',
    'm': 'objectivec',
    'mm': 'objectivec',
```

syntax-highlighter.ts

```
    'gradle': 'gradle',
    'groovy': 'groovy',
    'cmake': 'cmake',
    'tf': 'terraform',
    'vue': 'vue',
    'svelte': 'svelte',
    // Add more as needed
};

// --- Theme Mapping Logic ---

/**
 * Maps highlight.js CSS class names (found in `result.value`) to semantic token types
 * defined in the `SyntaxTheme` interface. This allows applying theme colors correctly.
 * @param className A space-separated string of CSS classes from highlight.js (e.g., "hljs-keyword", "hljs-string").
 * @returns The corresponding semantic token type key from `SyntaxTheme['tokenColors']`, or null if no specific mapping is found.
 */
function mapHljsClassToThemeToken(className: string): keyof SyntaxTheme['tokenColors'] | null {
    // Order matters slightly - more specific checks first if classes overlap
    if (className.includes('comment')) return 'comment';
    if (className.includes('keyword')) return 'keyword';
    if (className.includes('string')) return 'string';
    if (className.includes('number')) return 'number';
    if (className.includes('literal')) return 'literal'; // true, false, null
    if (className.includes('built_in')) return 'built_in'; // console, Math, standard library types/functions
    if (className.includes('function')) return 'function'; // Function definition keyword/name container
    if (className.includes('class') && className.includes('title')) return 'class'; // Class definition name
    // Title often applies to function names, class names (usage), important identifiers
    if (className.includes('title')) return 'title';
    if (className.includes('params')) return 'params'; // Function parameters
    if (className.includes('property')) return 'property'; // Object properties, member access
    if (className.includes('operator')) return 'operator';
    if (className.includes('punctuation')) return 'punctuation';
    if (className.includes('tag')) return 'tag'; // HTML/XML tags
    if (className.includes('attr') || className.includes('attribute')) return 'attr'; // HTML/XML attributes
    if (className.includes('variable')) return 'variable';
    if (className.includes('regexp')) return 'regexp';

    // Fallback if no specific class matched our defined types
    return null;
}

/**
 * Determines the font style for a token based on highlight.js classes and theme configuration.
 * @param className A space-separated string of CSS classes from highlight.js.
 * @param theme The active syntax theme configuration.
 * @returns The appropriate font style ('normal', 'italic', 'bold', 'bold-italic').
 */
function getFontStyle(className: string, theme: SyntaxTheme): HighlightedToken['fontStyle'] {
    const styles = theme.fontStyles || {};
    // Simple checks for now, could be expanded
```

syntax-highlighter.ts

```
    if (className.includes('comment') && styles.comment === 'italic') return 'italic';
    if (className.includes('keyword') && styles.keyword === 'bold') return 'bold';
    // Add more style mappings based on theme config if needed
    return 'normal'; // Default style
}


// --- Language Detection ---

/**
 * Detects the language identifier for syntax highlighting based on the file extension.
 * Uses the `LANGUAGE_MAP` for overrides, otherwise falls back to the extension itself.
 * @param extension The file extension (e.g., 'ts', 'py') without the leading dot.
 * @returns The language name recognized by highlight.js or the extension itself (lowercase).
 */
function detectLanguage(extension: string): string {
    const lowerExt = extension?.toLowerCase() || ''; // Handle potential null/undefined extension
    return LANGUAGE_MAP[lowerExt] || lowerExt; // Fallback to extension if no mapping
}

// --- HTML Parsing ---

/**
 * Parses the HTML output generated by highlight.js into an array of styled tokens.
 * This implementation uses a simple stack-based approach to handle nested spans
 * and correctly applies styles based on the active theme. It also decodes HTML entities.
 *
 * @param highlightedHtml The HTML string generated by `hljs.highlight().value`.
 * @param theme The syntax theme configuration object.
 * @returns An array of `HighlightedToken` objects representing the styled segments of the line.
 */
function parseHighlightedHtml(highlightedHtml: string, theme: SyntaxTheme): HighlightedToken[] {
    const tokens: HighlightedToken[] = [];
    // Stack to keep track of nested spans and their classes
    const stack: { tag: string; class?: string }[] = [];
    let currentText = '';
    let currentIndex = 0;

    while (currentIndex < highlightedHtml.length) {
        const tagStart = highlightedHtml.indexOf('<', currentIndex);

        // Extract text content occurring before the next tag (or until the end)
        const textBeforeTag = tagStart === -1
            ? highlightedHtml.substring(currentIndex)
            : highlightedHtml.substring(currentIndex, tagStart);

        if (textBeforeTag) {
            currentText += textBeforeTag;
        }

        // If no more tags, process remaining text and exit
        if (tagStart === -1) {
            if (currentText) {
                const decodedText = he.decode(currentText); // Decode entities
                const currentStyle = stack[stack.length - 1]; // Get style from top of stack
                const themeKey = currentStyle?.class ? mapHljsClassToThemeToken(currentStyle.class) : null;
```

syntax-highlighter.ts

```
              tokens.push({
                  text: decodedText,
                  color: themeKey ? (theme.tokenColors[themeKey] ?? theme.defaultColor) : theme.defaultColor,
                  fontStyle: currentStyle?.class ? getFontStyle(currentStyle.class, theme) : 'normal',
              });
          }
          break; // Exit loop
      }

      const tagEnd = highlightedHtml.indexOf('>', tagStart);
      if (tagEnd === -1) {
          // Malformed HTML (unclosed tag) - treat the rest as text
           logger.warn("Malformed HTML detected in highlighter output (unclosed tag).");
           currentText += highlightedHtml.substring(tagStart);
           // Process the potentially malformed remaining text
           if (currentText) {
               const decodedText = he.decode(currentText);
               const currentStyle = stack[stack.length - 1];
               const themeKey = currentStyle?.class ? mapHljsClassToThemeToken(currentStyle.class) : null;
               tokens.push({
                   text: decodedText,
                   color: themeKey ? (theme.tokenColors[themeKey] ?? theme.defaultColor) : theme.defaultColor,
                   fontStyle: currentStyle?.class ? getFontStyle(currentStyle.class, theme) : 'normal',
               });
           }
           break; // Exit loop
      }

      const tagContent = highlightedHtml.substring(tagStart + 1, tagEnd);
      const isClosingTag = tagContent.startsWith('/');

      // Process any accumulated text *before* handling the current tag
      if (currentText) {
           const decodedText = he.decode(currentText);
           const currentStyle = stack[stack.length - 1];
           const themeKey = currentStyle?.class ? mapHljsClassToThemeToken(currentStyle.class) : null;
           tokens.push({
               text: decodedText,
               color: themeKey ? (theme.tokenColors[themeKey] ?? theme.defaultColor) : theme.defaultColor, // Use default if key not in theme
               fontStyle: currentStyle?.class ? getFontStyle(currentStyle.class, theme) : 'normal',
           });
           currentText = ''; // Reset accumulated text
      }

      // Handle the tag itself
      if (isClosingTag) {
          // Closing tag: Pop the corresponding tag from the stack
          const tagName = tagContent.substring(1).trim();
          if (stack.length > 0 && stack[stack.length - 1].tag === tagName) {
              stack.pop();
          } else if (tagName === 'span') {
               // Allow potentially mismatched </span> tags from hljs sometimes? Log it.
```

syntax-highlighter.ts

```typescript
                 logger.debug(`Potentially mismatched closing tag </${tagName}> encountered.`);
                 if(stack.length > 0 && stack[stack.length - 1].tag === 'span') stack.pop(); // Try popping if top is span
            }
        } else {
            // Opening tag: Extract tag name and class, push onto stack
            // Improved regex to handle tags without attributes
            const parts = tagContent.match(/^([a-zA-Z0-9]+)(?:\s+(.*))?$/) || [null, tagContent, ''];
            const tagName = parts[1];
            const attributesStr = parts[2] || '';
            let className: string | undefined;
            // Simple class attribute parsing
            const classAttrMatch = attributesStr.match(/class="([^"]*)"/);
            if (classAttrMatch) {
                className = classAttrMatch[1];
            }
            stack.push({ tag: tagName, class: className });
        }

        // Move index past the processed tag
        currentIndex = tagEnd + 1;
    }

     // Filter out any tokens that ended up with empty text after decoding/parsing
    return tokens.filter(token => token.text.length > 0);
}


// --- Main Highlighting Function ---

/**
 * Applies syntax highlighting to the content of a single file.
 * It detects the language, processes the content line by line using highlight.js,
 * parses the resulting HTML into styled tokens, and applies colors/styles from the theme.
 * Includes fallbacks for unsupported languages or highlighting errors.
 *
 * @param fileInfo The `FileInfo` object containing the file's path, content, and extension.
 * @param theme The `SyntaxTheme` object defining the colors and styles to apply.
 * @returns A `HighlightedFile` object containing the original file info plus the array of `HighlightedLine` objects.
 */
export function highlightCode(fileInfo: FileInfo, theme: SyntaxTheme): HighlightedFile {
    const language = detectLanguage(fileInfo.extension);
    // Verify if the detected language is actually supported by highlight.js
    const detectedLanguageName = hljs.getLanguage(language) ? language : 'plaintext';
    logger.debug(`Highlighting ${fileInfo.relativePath} as language: ${detectedLanguageName}`);

    const highlightedLines: HighlightedLine[] = [];
    // Robustly split lines, handling \n and \r\n
    const lines = fileInfo.content.split(/\r?\n/);

    try {
        // Process line by line
        lines.forEach((line, index) => {
            let lineTokens: HighlightedToken[];
            const lineNumber = index + 1; // 1-based line number

```

syntax-highlighter.ts

```
          if (line.trim() === '') {
              // Handle empty lines simply: one empty token
              lineTokens = [{ text: '', fontStyle: 'normal', color: theme.defaultColor }];
          } else {
              // *** REMOVED explicit type annotation for 'result' ***
              let result = null; // Initialize as null
              try {
                  // Attempt highlighting with the detected (and verified) language
                  if (detectedLanguageName !== 'plaintext') {
                      // ignoreIllegals helps prevent errors on slightly malformed code
                      result = hljs.highlight(line, { language: detectedLanguageName, ignoreIllegals: true });
                  } else {
                      // If language wasn't registered, try auto-detection as a fallback
                      logger.debug(`Attempting auto-detect for line ${lineNumber} in ${fileInfo.relativePath}`);
                      result = hljs.highlightAuto(line);
                  }
              } catch (highlightError) {
                  // Log specific highlighting errors but continue processing the file
                  logger.warn(`Highlighting failed for line ${lineNumber} in ${fileInfo.relativePath}, using plain text. Error: ${(highlightError as Error).message}`);
                  result = null; // Ensure result is null on error
              }

              // Parse the HTML output (or use encoded plain text as fallback)
              // Use optional chaining on result?.value
              const htmlToParse = result?.value ?? he.encode(line);
              lineTokens = parseHighlightedHtml(htmlToParse, theme);


// Final safety check: If parsing resulted in empty tokens for a non-empty line, use a single plain token
               if (lineTokens.length === 0 && line.length > 0) {
                  logger.debug(`Token parsing yielded empty array for non-empty line ${lineNumber} in ${fileInfo.relativePath}. Using plain text token.`);
                  lineTokens = [{ text: line, color: theme.defaultColor, fontStyle: 'normal' }];
              }
          }

          // Add the processed line (tokens) to the results
          highlightedLines.push({
              lineNumber: lineNumber,
              tokens: lineTokens,
          });
      });

  } catch (processingError) {
      // Catch unexpected errors during the line processing loop (less likely now)
      logger.error(`Critical error during highlighting loop for ${fileInfo.relativePath}: ${(processingError as Error).message}`);

// Fallback: return the file structure but with unhighlighted lines to prevent total failure
      const fallbackLines = lines.map((line, index) => ({
          lineNumber: index + 1,
          tokens: [{ text: line, color: theme.defaultColor, fontStyle: 'normal' as const }],
      }));
```

syntax-highlighter.ts

```
        return {
            ...fileInfo,
            language: 'plaintext', // Indicate highlighting failed
            highlightedLines: fallbackLines,
        };
    }

    // Return the processed file info with highlighted lines
    return {
        ...fileInfo,
        language: detectedLanguageName, // Store the language that was actually used for highlighting
        highlightedLines,
    };
}

```

utils/logger.ts

```typescript
/**
 * Defines the severity levels for log messages.
 */
export enum LogLevel {
    ERROR = 'ERROR',
    WARN = 'WARN',
    INFO = 'INFO',
    DEBUG = 'DEBUG',
    SUCCESS = 'SUCCESS'
}

/**
 * ANSI color codes for console output.
 */
const COLORS = {
    [LogLevel.ERROR]: '\x1b[31m', // Red
    [LogLevel.WARN]: '\x1b[33m', // Yellow
    [LogLevel.INFO]: '\x1b[36m', // Cyan
    [LogLevel.DEBUG]: '\x1b[35m', // Magenta
    [LogLevel.SUCCESS]: '\x1b[32m', // Green
    RESET: '\x1b[0m' // Reset color
};

/** Internal flag to control verbose output. */
let isVerbose = false;

/**
 * Sets the logging verbosity.
 * @param verbose If true, DEBUG level messages will be printed.
 */
export function setVerbose(verbose: boolean): void {
    isVerbose = !!verbose; // Ensure boolean value
    if (isVerbose) {
        // Use the log function itself to report verbose status
        log('Verbose logging enabled.', LogLevel.DEBUG);
    }
}

/**
 * Logs a message to the console with appropriate level and color.
 * DEBUG messages are only shown if verbose mode is enabled.
 * @param message The message string to log.
 * @param level The severity level of the message (defaults to INFO).
 */
export function log(message: string, level: LogLevel = LogLevel.INFO): void {
    // Skip DEBUG messages if not in verbose mode
    if (level === LogLevel.DEBUG && !isVerbose) {
        return;
    }

    const timestamp = new Date().toISOString();
    const color = COLORS[level] || COLORS.RESET;
    const reset = COLORS.RESET;

    // Construct the log string with timestamp, level, and message
    const logString = `${color}[${timestamp}] [${level}]${reset} ${message}`;

    // Use console.error for ERROR level, console.warn for WARN, console.log otherwise
    // This ensures logs go to the correct stream (stderr/stdout)
    switch (level) {
```

utils/logger.ts

```
        case LogLevel.ERROR:
            console.error(logString);
            break;
        case LogLevel.WARN:
            console.warn(logString);
            break;
        default:
            console.log(logString);
            break;
    }
}

/**
 * A convenient wrapper object for logging functions by level.
 */
export const logger = {
    error: (message: string) => log(message, LogLevel.ERROR),
    warn: (message: string) => log(message, LogLevel.WARN),
    info: (message: string) => log(message, LogLevel.INFO),
    debug: (message: string) => log(message, LogLevel.DEBUG),
    success: (message: string) => log(message, LogLevel.SUCCESS),
    setVerbose: setVerbose,
    /** Checks if verbose logging is currently enabled. */
    isVerbose: (): boolean => isVerbose,
};


```

utils/themes.ts

```
import { SyntaxTheme } from './types';

/**
 * Defines the 'light' syntax highlighting theme, similar to GitHub's light theme.
 */
const lightTheme: SyntaxTheme = {
    defaultColor: '#24292e', // Default text color
    backgroundColor: '#ffffff', // White background for code blocks
    lineNumberColor: '#aaaaaa', // Light gray for line numbers
    lineNumberBackground: '#f6f8fa', // Very light gray background for the line number gutter
    headerFooterColor: '#586069', // Medium gray for text in headers/footers
    headerFooterBackground: '#f6f8fa', // Match line number background for consistency
    borderColor: '#e1e4e8', // Light gray border color for separators and containers
    tokenColors: {
        comment: '#6a737d',    // Gray
        keyword: '#d73a49',    // Red
        string: '#032f62',     // Dark blue
        number: '#005cc5',     // Blue
        literal: '#005cc5',    // Blue (true, false, null)
        built_in: '#005cc5',   // Blue (console, Math, etc.)
        function: '#6f42c1',   // Purple (function definitions)
        title: '#6f42c1',      // Purple (function/class usage, important identifiers)
        class: '#6f42c1',      // Purple (class definitions)
        params: '#24292e',     // Default text color for parameters
        property: '#005cc5',   // Blue for object properties/member access
        operator: '#d73a49',   // Red
        punctuation: '#24292e',// Default text color
        tag: '#22863a',        // Green (HTML/XML tags)
        attr: '#6f42c1',       // Purple (HTML/XML attributes)
        variable: '#e36209',   // Orange (variables)
        regexp: '#032f62',     // Dark blue
    },
    fontStyles: {
        comment: 'italic',
    }
};

/**
 * Defines the 'dark' syntax highlighting theme, similar to GitHub's dark theme.
 */
const darkTheme: SyntaxTheme = {
    defaultColor: '#c9d1d9', // Light gray default text
    backgroundColor: '#0d1117', // Very dark background for code blocks
    lineNumberColor: '#8b949e', // Medium gray for line numbers
    lineNumberBackground: '#161b22', // Slightly lighter dark background for the gutter
    headerFooterColor: '#8b949e', // Medium gray for text in headers/footers
    headerFooterBackground: '#161b22', // Match line number background
    borderColor: '#30363d', // Darker gray border color
    tokenColors: {
        comment: '#8b949e',    // Medium gray
        keyword: '#ff7b72',    // Light red/coral
        string: '#a5d6ff',     // Light blue
        number: '#79c0ff',     // Bright blue
        literal: '#79c0ff',    // Bright blue
        built_in: '#79c0ff',   // Bright blue
        function: '#d2a8ff',   // Light purple
        title: '#d2a8ff',      // Light purple
        class: '#d2a8ff',      // Light purple
        params: '#c9d1d9',     // Default text color
        property: '#79c0ff',   // Bright blue
```

utils/themes.ts

```
        operator: '#ff7b72',    // Light red/coral
        punctuation: '#c9d1d9',// Default text color
        tag: '#7ee787',         // Light green
        attr: '#d2a8ff',        // Light purple
        variable: '#ffa657',    // Light orange
        regexp: '#a5d6ff',      // Light blue
    },
    fontStyles: {
        comment: 'italic',
    }
};

// Add more themes here following the SyntaxTheme interface
// e.g., const solarizedLightTheme: SyntaxTheme = { ... };

/**
 * A record mapping theme names (lowercase) to their corresponding SyntaxTheme objects.
 * Used to look up themes based on the command-line option.
 */
export const themes: Record<string, SyntaxTheme> = {
    light: lightTheme,
    dark: darkTheme,
    // Add other themes here:
    // solarized: solarizedLightTheme,
};

/**
 * Retrieves the theme object for a given theme name.
 * Falls back to the 'light' theme if the requested theme name is not found.
 * @param themeName The name of the theme requested (case-insensitive).
 * @returns The corresponding SyntaxTheme object.
 */
export function getTheme(themeName: string): SyntaxTheme {
    // Normalize the input name (lowercase, default to 'light' if null/undefined)
    const normalizedName = themeName?.toLowerCase() || 'light';
    const theme = themes[normalizedName];

    // Check if the theme exists
    if (!theme) {
        // Log a warning if the theme wasn't found and we're falling back
        console.warn(`[Theme Warning] Theme "${themeName}" not found. Available themes: ${
->  Object.keys(themes).join(', ')}. Falling back to "light" theme.`);
        return themes.light; // Return the default light theme
    }
    return theme; // Return the found theme
}
```

utils/types.ts

```typescript
/**
 * Represents information about a single file discovered within the target repository.
 * This interface holds metadata and the raw content before processing.
 */
export interface FileInfo {
  /** The absolute path to the file on the filesystem. */
  absolutePath: string;

  /** The path to the file relative to the root of the scanned repository. Used for display and TOC generation. */
  relativePath: string;
  /** The raw text content of the file, read as UTF-8. */
  content: string;

  /** The file extension (e.g., 'ts', 'js', 'py') without the leading dot, converted to lowercase. */
  extension: string;

  /** The programming language detected for syntax highlighting purposes. Initially empty, populated by the highlighter. */
  language: string;
}

/**
 * Represents a single, styled segment (token) within a line of highlighted code.
 * Tokens are typically keywords, strings, comments, operators, etc.
 */
export interface HighlightedToken {
  /** The text content of this specific token. */
  text: string;

  /** Optional: The hex color code (e.g., '#0000ff') determined by the syntax theme for this token type. */
  color?: string;

  /** Optional: The font style ('normal', 'italic', 'bold', 'bold-italic') determined by the syntax theme. Defaults to 'normal'. */
  fontStyle?: 'normal' | 'italic' | 'bold' | 'bold-italic';
}

/**
 * Represents a single line of source code after syntax highlighting,
 * broken down into styled tokens.
 */
export interface HighlightedLine {
  /** The original line number (1-based) in the source file. */
  lineNumber: number;
  /** An array of styled tokens that make up this line. */
  tokens: HighlightedToken[];
}

/**
 * Represents a file after its content has been processed by the syntax highlighter.
 * Extends FileInfo with the tokenized lines.
 */
export interface HighlightedFile extends FileInfo {
  /** An array of highlighted lines, each containing styled tokens. */
  highlightedLines: HighlightedLine[];
}

```

utils/types.ts

```typescript
/**
 * Configuration options controlling the PDF generation process.
 * These are typically derived from command-line arguments or defaults.
 */
export interface PdfOptions {
  /** The absolute path where the output PDF file will be saved. */
  output: string;
  /** The main title displayed on the cover page of the PDF document. */
  title: string;
  /** The font size (in points) to use for rendering code blocks. */
  fontSize: number;
  /** Flag indicating whether line numbers should be displayed next to the code. */
  showLineNumbers: boolean;
  /** The identifier (e.g., 'light', 'dark') of the syntax highlighting theme to use. */
  theme: string;
  /**
   * The paper size for the PDF document. Can be a standard name ('A4', 'Letter')
   * or a custom size specified as [width, height] in PDF points (72 points per inch).
   */
  paperSize: 'A4' | 'Letter' | [number, number];
  /** Margins (in points) for the top, right, bottom, and left edges of each page. */
  margins: { top: number; right: number; bottom: number; left: number };
  /** The height (in points) reserved for the header section on each code page. */
  headerHeight: number;
  /** The height (in points) reserved for the footer section on each code page. */
  footerHeight: number;
  /** The title text used for the Table of Contents page. */
  tocTitle: string;

  /** The name of the font to use for rendering code blocks (e.g., 'Courier', 'Consolas'). Must be a standard PDF font or embedded. */
  codeFont: string;

  /** The name of the font to use for non-code text (titles, TOC, headers, footers) (e.g., 'Helvetica', 'Times-Roman'). Must be a standard PDF font or embedded. */
  textFont: string;
}

/**
 * Defines the color scheme and styling rules for a syntax highlighting theme.
 * Used by the PDF renderer to apply colors and styles to code tokens.
 */
export interface SyntaxTheme {
  /** The default text color used when no specific token rule applies. */
  defaultColor: string;
  /** The background color for the main code rendering area. */
  backgroundColor: string;
  /** The text color for line numbers. */
  lineNumberColor: string;
  /** The background color for the line number gutter area. */
  lineNumberBackground: string;
  /** The text color used in page headers and footers. */
  headerFooterColor: string;
  /** The background color used for page headers and footers. */
  headerFooterBackground: string;

  /** The color used for border lines (e.g., around code blocks, header/footer separators). */
  borderColor: string;

```

utils/types.ts

```
  /** A mapping of semantic token types (derived from highlight.js classes) to specific hex c
  olor codes. */
  tokenColors: {
    keyword?: string;
    string?: string;
    comment?: string;
    number?: string;
    function?: string; // e.g., function name definition
    class?: string;    // e.g., class name definition
    title?: string;    // e.g., function/class usage, important identifiers
    params?: string;   // Function parameters
    built_in?: string; // Built-in functions/variables/types
    literal?: string;  // e.g., true, false, null, undefined
    property?: string; // Object properties, member access
    operator?: string;
    punctuation?: string;
    attr?: string;     // HTML/XML attributes names
    tag?: string;      // HTML/XML tags names including </>
    variable?: string; // Variable declarations/usage
    regexp?: string;   // Regular expressions
    // Add more specific highlight.js scopes as needed (e.g., 'meta', 'section', 'type')
  };
  /** Optional: A mapping of semantic token types to specific font styles. */
  fontStyles?: {
    comment?: 'italic';
    keyword?: 'bold';
    // Add more styles if desired
  };
}
```