Code Repository Documentation

Repository: src

Generated: 4/30/2025, 1:04:15 AM

Table of Contents

cli.ts

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

import { Command } 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';

const program = new Command();

// Get version from package.json
const packageJsonPath = path.join(__dirname, '..', 'package.json'); // Adjust path if needed
const packageJson = fs.readJsonSync(packageJsonPath);

program
    .name('xprinto')
    .description('Convert code repositories to beautiful PDFs with syntax highlighting.')
    .version(packageJson.version)
    .argument('<repository-path>', 'Path to the code repository or directory')
    .option('-o, --output <path>', 'Output path for the generated PDF file', 'code-output.pdf')
    .option('-t, --title <title>', 'Title for the PDF document', 'Code Repository Documentation')
    .option('-f, --font-size <size>', 'Font size for code blocks', '9') // Adjusted default
    .option('--theme <name>', `Syntax highlighting theme (available: ${Object.keys(themes).join(', ')}
  )`, 'light')
    .option('--line-numbers', 'Show line numbers in code blocks (default)')
    .option('--no-line-numbers', 'Hide line numbers in code blocks')
    .option('--paper-size <size>', 'Paper size (A4, Letter, or width,height in points)', 'A4')
    .option('-v, --verbose', 'Enable verbose logging output', false)
    .action(async (repoPathArg, options) => {
        logger.setVerbose(options.verbose);
        // Set env var for verbose stack traces in main
        if (options.verbose) {
            process.env.XP_VERBOSE = 'true';
        }

        const resolvedRepoPath = path.resolve(repoPathArg);
        const resolvedOutputPath = path.resolve(options.output);

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

        // Validate input path exists and is a directory
        try {
            const stats = await fs.stat(resolvedRepoPath);
            if (!stats.isDirectory()) {
                logger.error(`Input path must be a directory: ${resolvedRepoPath}`);
                process.exit(1);
            }
        } catch (error) {
            logger.error(`Cannot access input path: ${resolvedRepoPath}`);
            logger.error((error as Error).message);
            process.exit(1);
        }

        // Validate theme
        if (!themes[options.theme.toLowerCase()]) {
             logger.error(`Invalid theme specified: ${options.theme}. Available: ${Object
  .keys(themes).join(', ')}`);
```

cli.ts

```
            process.exit(1);
        }

        // Parse paper size
        let paperSizeOption: PdfOptions['paperSize'];
        if (options.paperSize.includes(',')) {
            const dims = options.paperSize.split(',').map(Number);
            if (dims.length === 2 && !isNaN(dims[0]) && !isNaN(dims[1]) && dims[0] > 0 && dims[1] > 0
  ) {
                paperSizeOption = [dims[0], dims[1]];
            } else {
                logger.error(
  'Invalid paper size format. Use "width,height" in points (e.g., "595,842").');
                process.exit(1);
            }
        } else if (options.paperSize.toUpperCase() === 'A4' || options.paperSize.toUpperCase() ===
  'LETTER') {
            paperSizeOption = options.paperSize.toUpperCase() as 'A4' | 'Letter';
        } else {
            logger.error('Invalid paper size. Use "A4", "Letter", or "width,height".');
            process.exit(1);
        }


        // Construct PDF options object
        const pdfOptions: PdfOptions = {
            output: resolvedOutputPath,
            title: options.title,
            fontSize: parseInt(options.fontSize, 10),
            showLineNumbers: options.lineNumbers, // Commander handles --no- prefix automatically
            theme: options.theme.toLowerCase(),
            paperSize: paperSizeOption,
            // --- Sensible Defaults for Layout ---
            // Adjust these margins and heights as needed for aesthetics
            margins: { top: 50, right: 40, bottom: 50, left: 40 },
            headerHeight: 25, // Space for file path header
            footerHeight: 25, // Space for page number footer
            tocTitle: "Table of Contents",
            codeFont: 'Courier', // Standard monospace PDF font
            textFont: 'Helvetica' // Standard sans-serif PDF font
        };

         // Validate font size
         if (isNaN(pdfOptions.fontSize) || pdfOptions.fontSize <= 0) {
            logger.error(`Invalid font size: ${options.fontSize}. Must be a positive number.`);
            process.exit(1);
        }


        // Run the main logic
        await run(resolvedRepoPath, pdfOptions);
    });

// Make sure to parse arguments
program.parse(process.argv);
```

file-finder.ts

```
import path from 'path';
import fs from 'fs-extra'; // Using fs-extra for convenience like pathExists
import { glob } from 'glob';
import ignore, { Ignore } from 'ignore';
import { logger } from './utils/logger';
import { FileInfo } from './utils/types';

// List of common binary file extensions (can be expanded)
const BINARY_EXTENSIONS = new Set([
    'png', 'jpg', 'jpeg', 'gif', 'bmp', 'tiff', 'webp',
    'mp3', 'wav', 'ogg', 'flac',
    'mp4', 'avi', 'mov', 'wmv', 'mkv',
    'pdf', 'doc', 'docx', 'xls', 'xlsx', 'ppt', 'pptx',
    'zip', 'rar', 'gz', 'tar', '7z',
    'exe', 'dll', 'so', 'dylib', 'app',
    'o', 'a', 'obj',
    'jar', 'class',
    'pyc',
    'lock', // Lock files
    'log', // Log files often not needed
    'svg', // Sometimes treated as code, sometimes as binary asset
    // Add more as needed
]);

// Files to always ignore regardless of .gitignore
const ALWAYS_IGNORE = [
    '**/node_modules/**',
    '**/.git/**',
    '**/.svn/**',
    '**/.hg/**',
    '**/.vscode/**',
    '**/.idea/**',
    '**/dist/**', // Common build output directory
    '**/build/**', // Common build output directory
    '**/coverage/**', // Coverage reports
];

/**
 * Finds relevant code files in a directory, respecting .gitignore.
 * @param repoPath The absolute path to the repository root.
 * @returns A promise resolving to an array of FileInfo objects.
 */
export async function findCodeFiles(repoPath: string): Promise<FileInfo[]> {
    logger.info(`Scanning directory: ${repoPath}`);

    // 1. Initialize ignore instance and add always-ignored patterns
    const ig = ignore().add(ALWAYS_IGNORE);

    // 2. Find and load .gitignore files
    const gitignoreFiles = await glob('**/.gitignore', {
        cwd: repoPath,
        absolute: true,
        dot: true, // Include dotfiles like .gitignore
        ignore: ['**/node_modules/**', '**/.git/**'], // Avoid searching in these
    });

    for (const gitignorePath of gitignoreFiles) {
        try {
            if (await fs.pathExists(gitignorePath)) {
                const content = await fs.readFile(gitignorePath, 'utf-8');
```

file-finder.ts

```
                const relativeDir = path.dirname(path.relative(repoPath, gitignorePath));
                // Add patterns relative to the .gitignore file's location
                ig.add(content.split(/\r?\n/).map(line => {
                    // Handle patterns relative to the .gitignore location
                    if (line.trim() && !line.startsWith('#')) {
                        // If pattern doesn't start with '/', make it relative to the dir
                        if (!line.startsWith('/') && relativeDir !== '.') {

// Prepend directory path if pattern is not absolute within gitignore context
                            if (!line.startsWith('!')) { // Handle negations separately
                                return path.join(relativeDir, line).replace(/\\/g, '/');
                            } else {
                                // For negated patterns, keep them relative but adjust path
                                return '!' + path.join(relativeDir, line.substring(1)).replace(/\\/g,
'/');
                            }
                        }
                        return line;
                    }
                    return ''; // Ignore empty lines/comments
                }).filter(Boolean)); // Filter out empty strings
                logger.debug(`Loaded .gitignore: ${gitignorePath}`);
            }
        } catch (error) {
            logger.warn(`Failed to read or parse .gitignore file ${gitignorePath}: ${(error as Error
).message}`);
        }
    }

    // 3. Find all files using glob initially (excluding directories)
    const allFiles = await glob('**/*', {
        cwd: repoPath,
        absolute: true,
        nodir: true, // Only files, not directories
        dot: true, // Include dotfiles (like .eslintrc, .prettierrc)
        follow: false, // Don't follow symlinks to avoid potential loops/issues
        ignore: ['**/node_modules/**', '**/.git/**'], // Basic ignore for performance
    });

    logger.info(`Found ${allFiles.length} total files initially.`);

    // 4. Filter files
    const includedFiles: FileInfo[] = [];
    for (const absolutePath of allFiles) {
        const relativePath = path.relative(repoPath, absolutePath).replace(/\\/g, '/');
// Use forward slashes

        // Skip if ignored by .gitignore rules or always-ignore list
        if (ig.ignores(relativePath)) {
            logger.debug(`Ignoring (gitignore): ${relativePath}`);
            continue;
        }

        // 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}`);
            continue;
        }
```

file-finder.ts

```
        // Skip potentially very large files (e.g., > 5MB) - adjust as needed
        try {
            const stats = await fs.stat(absolutePath);
            if (stats.size > 5 * 1024 * 1024) {
                logger.warn(`Ignoring (large file > 5MB): ${relativePath}`);
                continue;
            }
        } catch (error) {
            logger.warn(`Could not get stats for ${relativePath}: ${(error as Error).message}`);
            continue; // Skip if stats fail
        }

        // If we reach here, include the file
        try {
            const content = await fs.readFile(absolutePath, 'utf-8');
            // Basic check for binary content (presence of null bytes) - might need refinement
             if (content.includes('\u0000')) {
                 logger.debug(`Ignoring (likely binary content): ${relativePath}`);
                 continue;
             }

            includedFiles.push({
                absolutePath,
                relativePath,
                content,
                extension,
                language: '', // Language will be detected later
            });
        } catch (error) {
            // Might fail if it's not UTF-8, likely binary
            logger.warn(`Could not read file ${relativePath} as UTF-8 (skipping): ${(error as Error
  ).message}`);
        }
    }

    logger.success(`Found ${includedFiles.length} relevant code files to include.`);
    return includedFiles;
}
```

main.ts

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

/**
 * Main orchestration function for the xprinto tool.
 *
 * @param repoPath Absolute path to the repository/directory.
 * @param options PDF generation options from the CLI.
 */
export async function run(repoPath: string, options: PdfOptions): Promise<void> {
    try {
        logger.info(`Processing repository: ${repoPath}`);
        logger.info(`Output PDF: ${options.output}`);
        logger.info(`Theme: ${options.theme}, Font Size: ${options.fontSize}, Line Numbers:
  ${options.showLineNumbers}`);

        // 1. Find relevant code files
        const filesToProcess = await findCodeFiles(repoPath);

        if (filesToProcess.length === 0) {
            logger.warn("No relevant code files found to process in the specified path.");
            return;
        }

        // 2. Load the selected theme
        const theme = getTheme(options.theme);
        logger.info(`Using theme: ${options.theme}`);

        // 3. Highlight code for each file
        logger.info("Applying syntax highlighting...");
        const highlightedFiles: HighlightedFile[] = filesToProcess.map(fileInfo => {
            return highlightCode(fileInfo, theme);
        });
        logger.info("Syntax highlighting complete.");


        // 4. Generate the PDF
        logger.info("Generating PDF document...");
        const repoName = path.basename(repoPath); // Use directory name for cover page
        await generatePdf(highlightedFiles, options, theme, repoName);

    } catch (error) {
        logger.error(`An unexpected error occurred: ${(error as Error).message}`);
        // Log stack trace in verbose mode
        if (process.env.XP_VERBOSE === 'true') { // Check env var set by CLI perhaps
             console.error((error as Error).stack);
        }
        // Ensure the process exits with an error code if run from CLI
        process.exitCode = 1;
    }
}
```

pdf-renderer.ts

```typescript
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;
const DEFAULT_LINE_HEIGHT_MULTIPLIER = 1.4; // Adjust for code readability
const TOC_INDENT = 20; // Points to indent file names under directories in TOC
const WRAP_INDENT_MULTIPLIER = 2; // How many characters to indent wrapped lines
const TOC_DOT_PADDING = 5; // Points padding around dots
const CODE_BLOCK_PADDING = 10; // Padding inside the code block container

// --- Helper Functions ---

/**
 * Converts paper size name to points array or returns the array.
 */
function getPaperSizeInPoints(size: PdfOptions['paperSize']): [number, number] {
    if (Array.isArray(size)) {
        return size;
    }
    switch (size.toUpperCase()) {
        case 'LETTER':
            return [8.5 * POINTS_PER_INCH, 11 * POINTS_PER_INCH];
        case 'A4':
        default:
            return [595.28, 841.89]; // A4 dimensions in points
    }
}

/**
 * Calculates the available content height on a page.
 */
function getContentHeight(doc: PDFKit.PDFDocument, options: PdfOptions): number {
    const pageHeight = doc.page.height;
    return pageHeight - options.margins.top - options.margins.bottom - options.headerHeight - options.
  footerHeight;
}

/**
 * Calculates the available content width on a page.
 */
 function getContentWidth(doc: PDFKit.PDFDocument, options: PdfOptions): number {
    const pageWidth = doc.page.width;
    return pageWidth - options.margins.left - options.margins.right;
}


// --- PDF Rendering Sections ---

/**
 * Adds a cover page to the PDF document.
 */
function addCoverPage(doc: PDFKit.PDFDocument, options: PdfOptions, repoName: string): void {
    // Ensure we always add a page, even if it's the very first one
    doc.addPage();
    const contentWidth = getContentWidth(doc, options);
```

pdf-renderer.ts

```
    const contentHeight = getContentHeight(doc, options); // Use full page height for cover
    const centerX = doc.page.margins.left + contentWidth / 2;

    // Title
    doc.font(options.textFont + '-Bold')
       .fontSize(24)
       .text(options.title, doc.page.margins.left, doc.page.margins.top + contentHeight * 0.2, { align
  : 'center', width: contentWidth });

    doc.moveDown(2);

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

    doc.moveDown(1);

    // Generation Date
    doc.fontSize(12)
       .fillColor('#555555') // Use a less prominent color
       .text(`Generated: ${new Date().toLocaleString()}`, { align: 'center', width: contentWidth });

    logger.info('Added cover page.');
}

/**
 * Adds a Table of Contents page.
 */
function addTableOfContents(
    doc: PDFKit.PDFDocument,
    files: HighlightedFile[],
    options: PdfOptions,
    theme: SyntaxTheme,
    pageNumberOffset: number // Starting page number for files (after cover/TOC)
): Record<string, number> { // Returns map of relativePath to starting page number

    // Ensure we always add a page for the TOC
    doc.addPage();
    const contentWidth = getContentWidth(doc, options);
    const startY = doc.page.margins.top;
    doc.y = startY;

    // TOC Title
    doc.font(options.textFont + '-Bold')
       .fontSize(18)
       .fillColor(theme.defaultColor) // Use theme default color
       .text(options.tocTitle, { align: 'center', width: contentWidth });

    doc.moveDown(2);

    // 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, '/')}`;
        if (!filesByDir[dirKey]) filesByDir[dirKey] = [];
        filesByDir[dirKey].push(file);
    });
```

pdf-renderer.ts

```
    // Estimate page numbers BEFORE drawing TOC
    const pageEstimates: Record<string, number> = {}; // relativePath -> startPage
    let estimatedCurrentPage = pageNumberOffset;
    // Recalculate linesPerPage based on the current document's font size for TOC
    const tocLineHeight = 12 * 1.2; // Estimate TOC line height
    const tocLinesPerPage = Math.floor(getContentHeight(doc, options) / tocLineHeight);
    // Estimate pages needed for code files
    const codeLinesPerPage = Math.floor(getContentHeight(doc, options) / (options.fontSize *
  DEFAULT_LINE_HEIGHT_MULTIPLIER));


    const sortedDirs = Object.keys(filesByDir).sort();
    for (const dir of sortedDirs) {
        const sortedFiles = filesByDir[dir].sort((a, b) => a.relativePath.localeCompare(b.relativePath
  ));
        for (const file of sortedFiles) {
            pageEstimates[file.relativePath] = estimatedCurrentPage;
            const lineCount = file.highlightedLines.length;
            const estimatedPagesForFile = Math.max(1, Math.ceil(lineCount / codeLinesPerPage));
            estimatedCurrentPage += estimatedPagesForFile;
        }
    }
    logger.debug(`Estimated total pages (including cover/TOC): ${estimatedCurrentPage -1}`);


    // Render TOC using estimated page numbers
    doc.font(options.textFont).fontSize(12);
    const tocStartY = doc.y;
    const tocEndY = doc.page.height - options.margins.bottom;

    for (const dir of sortedDirs) {
        // Check if space is running out for directory header AND at least one file entry
        if (doc.y > tocEndY - (tocLineHeight * 2)) {
             doc.addPage();
             doc.y = doc.page.margins.top; // Reset Y to top margin
        }

        // Directory Header
        if (dir !== '/') {
            doc.moveDown(1);
            doc.font(options.textFont + '-Bold')
               .fillColor(theme.defaultColor) // Use theme default color
               .text(dir, { continued: false });
            doc.moveDown(0.5);
        }

        // Files in Directory
        const sortedFiles = filesByDir[dir].sort((a, b) => a.relativePath.localeCompare(b.relativePath
  ));
        for (const file of sortedFiles) {
             // Check for page break before file entry
             if (doc.y > tocEndY - tocLineHeight) {
                 doc.addPage();
                 doc.y = doc.page.margins.top; // Reset Y to top margin
             }

            const fileName = path.basename(file.relativePath);
            const pageNum = pageEstimates[file.relativePath]?.toString() || '?';
            const indent = (dir === '/') ? 0 : TOC_INDENT;
            const startX = doc.page.margins.left + indent;
```

pdf-renderer.ts

```
            const availableWidth = contentWidth - indent;
            const currentY = doc.y;

            // --- Calculate positions ---
            doc.font(options.textFont).fontSize(12).fillColor(theme.defaultColor);
            const nameWidth = doc.widthOfString(fileName);
            const pageNumWidth = doc.widthOfString(pageNum);

            const fileNameEndX = startX + nameWidth;
            const pageNumStartX = doc.page.margins.left + contentWidth - pageNumWidth;
  // Right align page number

            // --- Render file name ---
            doc.text(fileName, startX, currentY, {
                width: nameWidth, // Use measured width to prevent unwanted wrapping
                lineBreak: false,
                continued: false // Important: Don't continue after filename
            });

            // --- Render page number ---
            // Explicitly set the position for the page number
            doc.text(pageNum, pageNumStartX, currentY, {
                width: pageNumWidth,
                lineBreak: false,
                continued: false // Important: Don't continue after page number
            });

            // --- Render dots (if space allows) ---
            const dotsStartX = fileNameEndX + TOC_DOT_PADDING;
            const dotsEndX = pageNumStartX - TOC_DOT_PADDING;
            const dotsAvailableWidth = dotsEndX - dotsStartX;

            if (dotsAvailableWidth > doc.widthOfString('. ')) {
                const dot = '. ';
                const dotWidth = doc.widthOfString(dot);
                const numDots = Math.floor(dotsAvailableWidth / dotWidth);
                const dotsString = dot.repeat(numDots);

                doc.fillColor('#aaaaaa'); // Lighter color for dots
                // Draw dots at the correct Y position, between filename and page number
                doc.text(dotsString, dotsStartX, currentY, {
                    width: dotsAvailableWidth, // Use calculated width
                    lineBreak: false,
                    continued: false // Ensure this doesn't interfere
                });
            }

            // Move down AFTER rendering all parts of the line
            doc.moveDown(0.6);
        }
    }

    logger.info('Added Table of Contents.');
    return pageEstimates;
}

/**
 * Renders the header for a code page.
 */
function renderHeader(doc: PDFKit.PDFDocument, file: HighlightedFile, options: PdfOptions, theme:
```

pdf-renderer.ts

```
  SyntaxTheme ): void {
    const headerY = options.margins.top;
    const headerContentY = headerY + (options.headerHeight - 9) / 2; // Vertically center ~9pt text
    const contentWidth = getContentWidth(doc, options);
    const startX = options.margins.left;

    // Background
    doc.rect(startX, headerY, contentWidth, options.headerHeight)
       .fillColor(theme.headerFooterBackground)
       .fill();

    // File Path (truncated if too long)
    doc.font(options.textFont)
       .fontSize(9)
       .fillColor(theme.headerFooterColor)
       .text(file.relativePath, startX + CODE_BLOCK_PADDING, headerContentY, { // Use padding
           width: contentWidth - (CODE_BLOCK_PADDING * 2), // Adjust width for padding
           align: 'left',
           lineBreak: false,
           ellipsis: true
       });

    // Border line below header
    doc.moveTo(startX, headerY + options.headerHeight)
       .lineTo(startX + contentWidth, headerY + options.headerHeight)
       .lineWidth(0.5)
       .strokeColor(theme.borderColor)
       .stroke();
}

/**
 * Renders the footer for a code page.
 */
function renderFooter(doc: PDFKit.PDFDocument, currentPage: number, options: PdfOptions, theme:
  SyntaxTheme): void {
    const footerY = doc.page.height - options.margins.bottom - options.footerHeight;
    const footerContentY = footerY + (options.footerHeight - 9) / 2; // Vertically center ~9pt text
    const contentWidth = getContentWidth(doc, options);
    const startX = options.margins.left;

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

    // Page Number
    doc.font(options.textFont)
       .fontSize(9)
       .fillColor(theme.headerFooterColor)
       .text(`Page ${currentPage}`, startX, footerContentY, { // Use calculated Y
           width: contentWidth,
           align: 'center'
       });
}

/**
 * Renders the highlighted code for a file, handling line numbers, wrapping, and page breaks.
 */
```

pdf-renderer.ts

```
function renderCodeFile(
    doc: PDFKit.PDFDocument,
    file: HighlightedFile,
    options: PdfOptions,
    theme: SyntaxTheme,
    initialPageNumber: number // This is the LOGICAL page number this file starts on
): number { // Returns the last LOGICAL page number used by this file

    let currentPage = initialPageNumber; // Track the logical page number for the footer
    const contentWidth = getContentWidth(doc, options);
    const contentHeight = getContentHeight(doc, options);
    const startY = options.margins.top + options.headerHeight;
    const endY = doc.page.height - options.margins.bottom - options.footerHeight;
    const startX = options.margins.left;
    const lineHeight = options.fontSize * DEFAULT_LINE_HEIGHT_MULTIPLIER;

    // Calculate line number column width
    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) : 0;
    const lineNumberPaddingRight = 10; // Space between line number and code
    const codeStartX = startX + (options.showLineNumbers ? lineNumberWidth + lineNumberPaddingRight :
  CODE_BLOCK_PADDING);
    const codeWidth = contentWidth - (codeStartX - startX) - CODE_BLOCK_PADDING;
  // Subtract right padding
    const wrapIndent = ' '.repeat(WRAP_INDENT_MULTIPLIER);
    const wrapIndentWidth = doc.font(options.codeFont).fontSize(options.fontSize).widthOfString
  (wrapIndent);

    // --- Page Setup Function ---
    // This function now also returns the starting Y position for content on the page
    const setupPageVisuals = (): number => {
        renderHeader(doc, file, options, theme);
        renderFooter(doc, currentPage, options, theme); // Use the current logical page number
        const pageStartY = startY; // Top of the content area
        doc.y = pageStartY; // Reset Y position

        // --- Draw Code Block Container ---
        doc.rect(startX, pageStartY, contentWidth, contentHeight)
           .fillColor(theme.backgroundColor)
           .lineWidth(0.75)
           .strokeColor(theme.borderColor)
           .fillAndStroke();

        // Draw line number background and separator if shown
        if (options.showLineNumbers && lineNumberWidth > 0) { // Check width > 0
            doc.rect(startX, pageStartY, lineNumberWidth, contentHeight)
               .fillColor(theme.lineNumberBackground)
               .fill();
             doc.moveTo(startX + lineNumberWidth, pageStartY)
                .lineTo(startX + lineNumberWidth, pageStartY + contentHeight)
                .lineWidth(0.5)
                .strokeColor(theme.borderColor)
                .stroke();
        }
         // Return the Y position where content should start (after top padding)
         return pageStartY + CODE_BLOCK_PADDING / 2;
    };

    // --- Initial Page Setup ---
```

pdf-renderer.ts

```
    doc.addPage();
    let currentLineY = setupPageVisuals(); // Use the returned starting Y

    // --- Render Loop ---
    for (const line of file.highlightedLines) {
        const lineStartY = currentLineY; // Store the Y where this original line starts

        // Check if we need a new page BEFORE rendering the line
        // Use lineStartY for the check
        if (lineStartY + lineHeight > endY - CODE_BLOCK_PADDING) {
             doc.addPage();
             currentPage++; // Increment the logical page number for the footer
             currentLineY = setupPageVisuals(); // Set up visuals and get new starting Y
        }

        // 1. Draw Line Number (if enabled) - Use currentLineY
        if (options.showLineNumbers && lineNumberWidth > 0) {
            doc.font(options.codeFont)
               .fontSize(options.fontSize)
               .fillColor(theme.lineNumberColor)
               .text(
                   String(line.lineNumber).padStart(maxLineNumDigits, ' '),
                   startX + CODE_BLOCK_PADDING / 2, // Start drawing within padding
                   currentLineY, // Use the managed Y position
                   {
                       width: lineNumberWidth - CODE_BLOCK_PADDING, // Constrain width to padded area
                       align: 'right', // Right-align within the column
                       lineBreak: false
                   }
               );
        }

        // 2. Render Code Tokens (handling wrapping)
        let currentX = codeStartX; // Start code after line numbers/padding
        let isFirstTokenOfLine = true; // Flag for wrapping logic

        // Helper function to handle moving to the next line during wrapping
        const moveToNextWrapLine = () => {
            // Increment our managed Y position
            currentLineY += lineHeight;
            // Check for page break using the *new* Y position
            if (currentLineY + lineHeight > endY - CODE_BLOCK_PADDING) {
                doc.addPage();
                currentPage++; // Increment logical page number
                currentLineY = setupPageVisuals(); // Setup visuals and reset Y
            }
            // Set X for the wrapped line *after* potential page setup
            currentX = codeStartX + wrapIndentWidth; // Apply wrap indent for the new line
            // Draw wrap indicator if line numbers are shown - Use currentLineY
            if (options.showLineNumbers && lineNumberWidth > 0) {
                doc.font(options.codeFont).fontSize(options.fontSize).fillColor(theme.lineNumberColor)
                   .text('!ª,'startX + CODE_BLOCK_PADDING / 2, currentLineY, { width: lineNumberWidth -
  CODE_BLOCK_PADDING, align: 'right', lineBreak: false });
            }
        };


        // Iterate through tokens for the current source line
        for (const token of line.tokens) {
             doc.font(options.codeFont + (token.fontStyle === 'bold' ? '-Bold' : token.fontStyle ===
```

pdf-renderer.ts

```
'italic' ? '-Oblique' : ''))
              .fontSize(options.fontSize)
              .fillColor(token.color || theme.defaultColor);

          const tokenText = token.text;
          const tokenWidth = doc.widthOfString(tokenText);

          // Check if token fits on the current PDF line segment
          if (currentX + tokenWidth <= codeStartX + codeWidth) {
              // Fits: Draw it and update X - Use currentLineY
              doc.text(tokenText, currentX, currentLineY, { continued: true, lineBreak: false });
              currentX += tokenWidth;
          } else {
              // Needs wrapping: Process character by character or segment by segment
              let remainingText = tokenText;

              // Move to next line to start the wrapped segment
              // We need to handle the case where the *first* token overflows
              if (isFirstTokenOfLine && currentX === codeStartX) {
                  // First token overflows immediately, move before drawing anything
                  moveToNextWrapLine();
              } else if (!isFirstTokenOfLine) {
                   // Not the first token, move to start the wrap
                   moveToNextWrapLine();
              }

// If it's the first token but *some* part fit, the loop below handles subsequent moves.


              while (remainingText.length > 0) {
                  let fitsChars = 0;
                  let currentSegmentWidth = 0;
                  // Available width on the current (potentially wrapped) line
                  const availableWidth = (codeStartX + codeWidth) - currentX;

                  // Find how many characters fit
                  for (let i = 1; i <= remainingText.length; i++) {
                      const segment = remainingText.substring(0, i);
                      const width = doc.widthOfString(segment);
                      // Use a small tolerance to prevent issues with floating point comparisons
                      if (width <= availableWidth + 0.001) {
                          fitsChars = i;
                          currentSegmentWidth = width;
                      } else {
                          break; // Exceeded available width
                      }
                  }

                   if (fitsChars === 0 && remainingText.length > 0) {
                       // Cannot fit even one character
                       fitsChars = 1;
                       currentSegmentWidth = doc.widthOfString(remainingText[0]);
                       logger.warn(`Cannot fit character '${remainingText[0]}' on wrapped line
${line.lineNumber} of ${file.relativePath}.`);
                   }

                  const textToDraw = remainingText.substring(0, fitsChars);
                  // Draw the segment that fits - Use currentLineY
                  // Ensure font/color are set correctly for this segment
                  doc.font(options.codeFont + (token.fontStyle === 'bold' ? '-Bold' : token.fontStyle
```

pdf-renderer.ts

```
    === 'italic' ? '-Oblique' : ''))
                        .fontSize(options.fontSize)
                        .fillColor(token.color || theme.defaultColor);
                    doc.text(textToDraw, currentX, currentLineY, { continued: true, lineBreak: false
    });

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

                    // If there's more text in this token, move to the next line
                    if (remainingText.length > 0) {
                        moveToNextWrapLine();
                    }
                } // End while remainingText in token
            } // End else (wrapping needed)
             isFirstTokenOfLine = false; // After processing the first token, this flag is false
        } // End for loop (tokens)

        // ** Advance our managed Y position for the next source line **
        // This should happen regardless of wrapping.
        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
}


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

/**
 * Generates the PDF document from highlighted files.
 */
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();

    const doc = new PDFDocument({
        size: getPaperSizeInPoints(options.paperSize),
        margins: options.margins,
        autoFirstPage: false, // We explicitly add all pages
        bufferPages: true, // Recommended for complex layouts / page counting issues
        info: {
            Title: options.title,
            Author: 'xprinto',
            Creator: 'xprinto',
            CreationDate: new Date(),
        }
    });

    const outputDir = path.dirname(options.output);
    await fs.ensureDir(outputDir);
```

pdf-renderer.ts

```
    const writeStream = fs.createWriteStream(options.output);
    doc.pipe(writeStream);

    let physicalPageCount = 0; // Track actual pages added

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

    // 2. Table of Contents
    let tocPages = 0;
    // The logical page number where code files *should* start (after cover + TOC)
    let fileStartLogicalPageNumber = physicalPageCount + 1;

    if (files.length > 1) {
        const tocStartPhysicalPage = physicalPageCount + 1;
        // Pass the estimated logical start page for files to TOC for its calculations
        addTableOfContents(doc, files, options, theme, fileStartLogicalPageNumber); // Adds TOC page(s)
        const tocEndPhysicalPage = doc.bufferedPageRange().count;
        tocPages = tocEndPhysicalPage - physicalPageCount;
        physicalPageCount = tocEndPhysicalPage; // Update physical page count
        // Update the logical start page number for files *after* TOC is rendered
        fileStartLogicalPageNumber = physicalPageCount + 1;
        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).');
         // fileStartLogicalPageNumber remains physicalPageCount + 1
    }

    // 3. Render Code Files
    let lastLogicalPageNumber = physicalPageCount; // Track the logical page number for the footer

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

    for (const file of sortedFiles) {
        // Pass the correct starting logical page number for this file
        const currentFileStartLogicalPage = lastLogicalPageNumber + 1;
        logger.debug(`Rendering file: ${file.relativePath}, starting on logical page
  ${currentFileStartLogicalPage}`);
        // renderCodeFile returns the last logical page number used by that file
        lastLogicalPageNumber = renderCodeFile(doc, file, options, theme, currentFileStartLogicalPage);
    }

    // --- Finalize PDF ---
    doc.end();

    await new Promise<void>((resolve, reject) => {
        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();
        });
        writeStream.on('error', (err) => {
            logger.error(`Error writing PDF file: ${err.message}`);
            reject(err);
        });
    });
```

pdf-renderer.ts

```
}
```

syntax-highlighter.ts

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

// --- Language Mapping ---
// Add mappings for extensions highlight.js might not guess correctly
const LANGUAGE_MAP: Record<string, string> = {
    'ts': 'typescript',
    'tsx': 'typescript',
    'js': 'javascript',
    'jsx': 'javascript',
    'py': 'python',
    'rb': 'ruby',
    'java': 'java',
    'cs': 'csharp',
    'go': 'go',
    'php': 'php',
    'html': 'html',
    'css': 'css',
    'scss': 'scss',
    'less': 'less',
    'json': 'json',
    'yaml': 'yaml',
    'yml': 'yaml',
    'md': 'markdown',
    'sh': 'bash',
    'bash': 'bash',
    'zsh': 'bash',
    'sql': 'sql',
    'xml': 'xml',
    'kt': 'kotlin',
    'swift': 'swift',
    'pl': 'perl',
    'rs': 'rust',
    'lua': 'lua',
    'dockerfile': 'dockerfile',
    'h': 'c', // Often C or C++ header
    'hpp': 'cpp',
    'cpp': 'cpp',
    'c': 'c',
    'm': 'objectivec',
    'mm': 'objectivec',
    // Add more as needed
};

// --- Theme Mapping ---
// Maps highlight.js CSS classes to theme token types
function mapHljsClassToThemeToken(className: string): keyof SyntaxTheme['tokenColors'] | null {
    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
    if (className.includes('function')) return 'function'; // Function definition keyword
    // Check for title but exclude class titles specifically
    if (className.includes('title') && !className.includes('class')) return 'title';
  // Function/method names, important vars
```

syntax-highlighter.ts

```
    if (className.includes('class') && className.includes('title')) return 'class';
  // Class definition name
    if (className.includes('params')) return 'params'; // Function parameters
    if (className.includes('property')) return 'property'; // Object properties
    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';
    // Add more specific mappings if needed based on highlight.js output
    return null;
}

// --- Font Style Mapping ---
function getFontStyle(className: string, theme: SyntaxTheme): HighlightedToken['fontStyle'] {
    const styles = theme.fontStyles || {};
    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
    return 'normal'; // Return the literal 'normal'
}


/**
 * Detects the language for highlighting based on file extension.
 * @param extension The file extension (without the dot).
 * @returns The language name recognized by highlight.js or the extension itself.
 */
function detectLanguage(extension: string): string {
    const lowerExt = extension.toLowerCase();
    return LANGUAGE_MAP[lowerExt] || lowerExt; // Fallback to extension if no mapping
}

/**
 * Parses the HTML output of highlight.js to extract tokens with styles.
 * This version aims to be more robust in handling nested spans and plain text.
 * @param highlightedHtml The HTML string generated by hljs.highlight().value
 * @param theme The syntax theme to apply colors from.
 * @returns An array of HighlightedToken objects.
 */
function parseHighlightedHtml(highlightedHtml: string, theme: SyntaxTheme): HighlightedToken[] {
    const tokens: HighlightedToken[] = [];
    // Use a simple stack-based parser approach
    const stack: { tag: string; class?: string }[] = [];
    let currentText = '';
    let currentIndex = 0;

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

        // Text before the next tag (or end of string)
        const textBeforeTag = tagStart === -1
            ? highlightedHtml.substring(currentIndex)
            : highlightedHtml.substring(currentIndex, tagStart);

        if (textBeforeTag) {
            currentText += textBeforeTag;
        }
```

syntax-highlighter.ts

```
        if (tagStart === -1) {
            // End of string
            if (currentText) {
                const decodedText = he.decode(currentText); // Decode entities
                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,
                    fontStyle: currentStyle?.class ? getFontStyle(currentStyle.class, theme) : 'normal'
, // Use 'normal' literal
                });
            }
            break; // Exit loop
        }

        const tagEnd = highlightedHtml.indexOf('>', tagStart);
        if (tagEnd === -1) {
            // Malformed HTML? Treat rest as text
             logger.warn("Malformed HTML detected in highlighter output.");
             currentText += highlightedHtml.substring(tagStart);
             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,
                     fontStyle: currentStyle?.class ? getFontStyle(currentStyle.class, theme) :
'normal', // Use 'normal' literal
                 });
             }
             break;
        }

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

        // Process accumulated text before handling the tag
        if (currentText) {
             const decodedText = he.decode(currentText); // Decode entities just before pushing
             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,
                 fontStyle: currentStyle?.class ? getFontStyle(currentStyle.class, theme) : 'normal',
// Use 'normal' literal
             });
             currentText = ''; // Reset accumulated text
        }

        if (isClosingTag) {
            // Pop from stack if it's the corresponding closing tag
            const tagName = tagContent.substring(1).trim();
            if (stack.length > 0 && stack[stack.length - 1].tag === tagName) {
```

syntax-highlighter.ts

```
                stack.pop();
            } else {
                 logger.warn(`Mismatched closing tag </${tagName}> encountered.`);
            }
        } else {
            // Opening tag
            const parts = tagContent.split(/\s+/);
            const tagName = parts[0];
            let className: string | undefined;
            // Very basic class attribute parsing
            const classAttrMatch = tagContent.match(/class="([^"]*)"/);
            if (classAttrMatch) {
                className = classAttrMatch[1];
            }
            stack.push({ tag: tagName, class: className });
        }

        currentIndex = tagEnd + 1;
    }

     // Filter out empty tokens that might result from decoding/parsing artifacts
    return tokens.filter(token => token.text.length > 0);
}


/**
 * Highlights the code content of a file.
 * @param fileInfo Information about the file.
 * @param theme The syntax theme to use for colors.
 * @returns A HighlightedFile object with tokenized lines.
 */
export function highlightCode(fileInfo: FileInfo, theme: SyntaxTheme): HighlightedFile {
    const language = detectLanguage(fileInfo.extension);
    logger.debug(`Highlighting ${fileInfo.relativePath} as language: ${language}`);

    const highlightedLines: HighlightedLine[] = [];
    const lines = fileInfo.content.split(/\r?\n/); // Split into lines

    try {
        // Process line by line
        lines.forEach((line, index) => {
            let lineTokens: HighlightedToken[];

            if (line.trim() === '') {
                // Handle empty lines
                lineTokens = [{ text: '', fontStyle: 'normal', color: theme.defaultColor }];
  // Use 'normal' literal
            } else {
                // Define result type using the imported hljs object's types if needed,
                // but often type inference from the highlight functions is sufficient.
                // Using 'any' temporarily if inference fails or types are complex.
                let result: any;
  // Use 'any' or rely on inference. Avoid 'hljs.HighlightResult' directly.
                try {
                    // Try highlighting with the detected language
                    if (hljs.getLanguage(language)) {
                        result = hljs.highlight(line, { language: language, ignoreIllegals: true });
                    } else {
                        // Fallback to auto-detection if language is not supported
                        logger.debug(`Language '${language}
```

syntax-highlighter.ts

```
  ' not explicitly supported by highlight.js, attempting auto-detect for line ${index + 1} in ${fileInfo.relativePath}`);
                          result = hljs.highlightAuto(line);
                      }
                  } catch (e) {
                      logger.warn(`Highlighting failed for line ${index + 1} in ${fileInfo.relativePath}
  , using plain text. Error: ${(e as Error).message}`);
                      // Fallback: treat the whole line as default text
                      // Ensure the fallback structure matches HighlightResult structure minimally
                      result = { value: he.encode(line), language: 'plaintext', relevance: 0 };
  // Encode to mimic hljs output
                  }

                  // Parse the HTML output into tokens
                  // Ensure 'result.value' is a string before passing
                   lineTokens = parseHighlightedHtml(result?.value || he.encode(line), theme);


                   // If parsing results in empty tokens (shouldn't happen often), fallback
                   if (lineTokens.length === 0 && line.length > 0) {
                      logger.debug(`Token parsing resulted in empty array for non-empty line ${index + 1}
   in ${fileInfo.relativePath}. Using plain text token.`);
                      lineTokens = [{ text: line, color: theme.defaultColor, fontStyle: 'normal' }];
  // Use 'normal' literal
                  }
              }

              highlightedLines.push({
                  lineNumber: index + 1,
                  tokens: lineTokens,
              });
          });

    } catch (error) {
          logger.error(`Critical error during highlighting process for ${fileInfo.relativePath}:
  ${(error as Error).message}`);
          // Fallback: return unhighlighted structure
          const fallbackLines = lines.map((line, index) => ({
              lineNumber: index + 1,
              tokens: [{ text: line, color: theme.defaultColor, fontStyle: 'normal' as const }],
  // Use 'normal' literal and 'as const' for type safety
          }));
          return {
              ...fileInfo,
              language: 'plaintext', // Mark as plaintext due to error
              highlightedLines: fallbackLines, // This should now match the expected type
          };
    }

    return {
          ...fileInfo,
          language: language, // Store the detected language
          highlightedLines,
    };
}
```

utils/logger.ts

```typescript
// Simple console logger with levels and colors

export enum LogLevel {
    ERROR = 'ERROR',
    WARN = 'WARN',
    INFO = 'INFO',
    DEBUG = 'DEBUG',
    SUCCESS = 'SUCCESS'
}

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
};

let isVerbose = false;

export function setVerbose(verbose: boolean): void {
    isVerbose = verbose;
    if (isVerbose) {
        log('Verbose logging enabled.', LogLevel.DEBUG);
    }
}

export function log(message: string, level: LogLevel = LogLevel.INFO): void {
    if (level === LogLevel.DEBUG && !isVerbose) {
        return; // Don't log debug messages unless verbose is enabled
    }

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

    console.log(`${color}[${timestamp}] [${level}]${reset} ${message}`);

    // Optionally add more sophisticated logging here (e.g., to a file)
}

// Convenience functions
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
};
```

utils/themes.ts

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

// Define color themes here
// Using common hex color codes

const lightTheme: SyntaxTheme = {
    defaultColor: '#24292e', // GitHub default text
    backgroundColor: '#ffffff', // White background
    lineNumberColor: '#aaaaaa', // Light gray line numbers
    lineNumberBackground: '#f6f8fa', // Very light gray background for numbers
    headerFooterColor: '#586069', // Gray for header/footer text
    headerFooterBackground: '#f6f8fa', // Match line number background
    borderColor: '#e1e4e8', // Light border color
    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)
        class: '#6f42c1',      // Purple (class definitions)
        params: '#24292e',     // Default text color for params
        property: '#005cc5',   // Blue for object properties
        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',
    }
};

const darkTheme: SyntaxTheme = {
    defaultColor: '#c9d1d9', // Light gray default text
    backgroundColor: '#0d1117', // Very dark background
    lineNumberColor: '#8b949e', // Medium gray line numbers
    lineNumberBackground: '#161b22', // Slightly lighter dark background
    headerFooterColor: '#8b949e', // Medium gray for header/footer
    headerFooterBackground: '#161b22', // Match line number background
    borderColor: '#30363d', // Dark 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
        operator: '#ff7b72',   // Light red/coral
        punctuation: '#c9d1d9',// Default text color
        tag: '#7ee787',        // Light green
```

utils/themes.ts

```typescript
        attr: '#d2a8ff',        // Light purple
        variable: '#ffa657',    // Light orange
        regexp: '#a5d6ff',      // Light blue
    },
    fontStyles: {
        comment: 'italic',
    }
};

// Add more themes here (e.g., solarized, monokai)

export const themes: Record<string, SyntaxTheme> = {
    light: lightTheme,
    dark: darkTheme,
    // Add other themes here
};

export function getTheme(themeName: string): SyntaxTheme {
    return themes[themeName.toLowerCase()] || themes.light; // Default to light theme
}
```

utils/types.ts

```
/**
 * Represents information about a file found in the repository.
 */
export interface FileInfo {
    absolutePath: string; // Full path to the file
    relativePath: string; // Path relative to the repository root
    content: string;      // File content as a string
    extension: string;    // File extension (e.g., 'ts', 'js')
    language: string;     // Detected language for highlighting
  }

  /**
   * Represents a single token within a line of highlighted code.
   */
  export interface HighlightedToken {
    text: string;
    color?: string;     // Hex color code (e.g., '#0000ff')
    fontStyle?: 'normal' | 'italic' | 'bold' | 'bold-italic';
  }

  /**
   * Represents a single line of code with its tokens.
   */
  export interface HighlightedLine {
    lineNumber: number;
    tokens: HighlightedToken[];
  }

  /**
   * Represents a file with its content processed for highlighting.
   */
  export interface HighlightedFile extends FileInfo {
    highlightedLines: HighlightedLine[];
  }

  /**
   * Options for configuring the PDF generation process.
   */
  export interface PdfOptions {
    output: string;
    title: string;
    fontSize: number;
    showLineNumbers: boolean;
    theme: string; // Identifier for the theme (maps to colors)
    // Standard PDF page sizes (points)
    paperSize: 'A4' | 'Letter' | [number, number];
    margins: { top: number; right: number; bottom: number; left: number };
    headerHeight: number;
    footerHeight: number;
    tocTitle: string;
    codeFont: string; // Font for code blocks
    textFont: string; // Font for titles, TOC, headers/footers
  }

  /**
   * Defines the color scheme for a syntax highlighting theme.
   */
  export interface SyntaxTheme {
    defaultColor: string;
    backgroundColor: string; // Background for code blocks
```

utils/types.ts

```typescript
    lineNumberColor: string;
    lineNumberBackground: string;
    headerFooterColor: string;
    headerFooterBackground: string;
    borderColor: string;
    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
      literal?: string;  // e.g., true, false, null
      property?: string; // Object properties
      operator?: string;
      punctuation?: string;
      attr?: string;     // HTML/XML attributes
      tag?: string;      // HTML/XML tags
      variable?: string; // Variable declarations/usage
      regexp?: string;
      // Add more specific highlight.js scopes as needed
    };
    fontStyles?: { // Optional font styles
      comment?: 'italic';
      keyword?: 'bold';
      // Add more styles
    };
  }
```