Code Documentation

Source: /Users/jacob/research/xprinto/src

Generated on: 4/15/2025, 11:11:46 PM

This document contains a formatted representation of the code with syntax highlighting and line numbers for easy reference. Navigate through the document using the table of contents (if available).

Table of Contents

Note: Page numbers reflect actual document pagination.

index.ts

Language: TYPESCRIPT

```typescript
// Re-export necessary functions
export { generatePdfFromPath } from './pdf/generator';
export { readPath } from './file-reader';
export { highlightCode } from './syntax/highlighter';
export { log, LogLevel, setVerboseLogging } from './utils/logger';
```

file-reader.ts Language: TYPESCRIPT

```typescript
import fs from 'fs-extra';
import path from 'path';
import { glob } from 'glob'; // Updated import statement
import { log, LogLevel } from './utils/logger';

// Interface for file content
export interface FileInfo {
  path: string;
  relativePath: string;
  content: string;
  extension: string;
}

// Function to read a single file
export async function readFile(filePath: string, rootPath?: string): Promise<FileInfo> {
  // Read the raw content
  const rawContent = await fs.readFile(filePath, 'utf-8');

  // Ensure we're only processing actual source code
  // This prevents any system-level tags from being processed
  const content = rawContent
    .replace(/<[a-z_]+:[a-z_]+>[\s\S]*?<\/[a-z_]+:[a-z_]+>/g, '')
    .replace(/<[a-z_]+_[a-z_]+_from_[a-z_]+>[\s\S]*?<\/[a-z_]+_[a-z_]+_from_[a-z_]+>/g, '')
    .replace(/<[a-z_]+_[a-z_]+>[\s\S]*?<\/[a-z_]+_[a-z_]+>/g, '');

  const relativePath = rootPath ? path.relative(rootPath, filePath) : path.basename(filePath);
  const extension = path.extname(filePath).substring(1); // Remove the dot

  return {
    path: filePath,
    relativePath,
    content,
    extension
  };
}

// Function to read files from a directory recursively
export async function readDirectory(dirPath: string): Promise<FileInfo[]> {
  // Get all files in the directory and subdirectories - updated to use async/await wi
```

file-reader.ts Language: TYPESCRIPT

```
    th glob                                    !a
  const files = await glob('**/*.*', {
    cwd: dirPath,
    nodir: true,
    ignore: ['**/node_modules/**', '**/.git/**'] // Ignore node_modules and .git
  });

  log(span class="hljs-string">`Found ${files.length} files in directory ${dirPath}`
   /span>, LogLevel.INFO);                     !a

  // Read all files
  const fileInfos: FileInfo[] = [];
  for (const file of files) {
    const filePath = path.join(dirPath, file);
    try {
      const fileInfo = await readFile(filePath, dirPath);
      fileInfos.push(fileInfo);
    } catch (err) {
      log(span class="hljs-string">`Error reading file ${filePath}:
   span class="hljs-subst">${(err as Error).message}/span>`/span>, LogLevel.ERROR);
    }
  }

  return fileInfos;
}

// Function to read from either a file or directory
export async function readPath(span class="hljs-params">inputPath: string/span>):
   PromiseFileInfo[]> {                        !a
  const stats = await fs.stat(inputPath);

  if (stats.isFile()) {
    const fileInfo = await readFile(inputPath);
    return [fileInfo];
  } else if (stats.isDirectory()) {
    return readDirectory(inputPath);
  } else {
    throw new Error(span class="hljs-string">`Invalid path: ${inputPath}
     is neither a file nor a directory`/span>););
  }
}
```

cli.ts Language: TYPESCRIPT

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

import { Command } from 'commander';
import path from 'path';
import fs from 'fs-extra';
import { generatePdfFromPath } from './pdf/generator';
import { log, LogLevel } from './utils/logger';

const program = new Command();

program
  .name('xprinto')
  .description('Convert code to beautiful PDFs with syntax highlighting')
  .version('1.0.0')
  .argument(span class="hljs-string">'path>'/span>,
   'File or directory path to convert')
  .option(span class="hljs-string">'-o, --output path>'/span>,
   'Output path for the PDF', './output.pdf')
  .option(span class="hljs-string">'-t, --title title>'/span>,
   'Title for the PDF document', 'Code Documentation')
  .option(span class="hljs-string">'--theme theme>'/span>, 'Syntax highlighting theme'
   , 'github')
  .option(span class="hljs-string">'--font-size size>'/span>, 'Font size for code',
   '8')
  .option('--line-numbers', 'Show line numbers', true)
  .option('--no-line-numbers', 'Hide line numbers')
  .option('-v, --verbose', 'Enable verbose logging')
  .action(async (inputPath: string, options) => {
    try {
      // Set log level based on verbose flag
      if (options.verbose) {
        log('Verbose mode enabled', LogLevel.INFO);
      }

      // Resolve input path
      const resolvedPath = path.resolve(inputPath);

      // Check if path exists
      if (!fs.existsSync(resolvedPath)) {
        log(span class="hljs-string">`Path does not exist: ${resolvedPath}`/span>,
   LogLevel.ERROR);
        process.exit(1);
      }

      // Generate PDF
```

cli.ts Language: TYPESCRIPT

```
40        log(span class="hljs-string">`Converting ${resolvedPath} to PDF...`/span>,
    LogLevel.INFO);
41        await generatePdfFromPath(
42          resolvedPath,
43          options.output,
44          {
45            title: options.title,
46            theme: options.theme,
47            fontSize: parseInt(options.fontSize, 10),
48            showLineNumbers: options.lineNumbers
49          }
50        );
51
52        log(span class="hljs-string">`PDF generated successfully: ${options.output}`
    /span>, LogLevel.SUCCESS);
53      } catch (err) {
54        log(span class="hljs-string">`Error: span class="hljs-subst">${(err as Error
    ).message}/span>`/span>, LogLevel.ERROR);
55        process.exit(1);
56      }
57    });
58
59  program.parse();
```

utils/logger.ts Language: TYPESCRIPT

```typescript
export enum LogLevel {
    ERROR = 'ERROR',
    WARNING = 'WARNING',
    INFO = 'INFO',
    SUCCESS = 'SUCCESS',
    DEBUG = 'DEBUG'
  }

  const COLORS = {
    [LogLevel.ERROR]: '\x1b[31m', // Red
    [LogLevel.WARNING]: '\x1b[33m', // Yellow
    [LogLevel.INFO]: '\x1b[36m', // Cyan
    [LogLevel.SUCCESS]: '\x1b[32m', // Green
    [LogLevel.DEBUG]: '\x1b[35m', // Magenta
    RESET: '\x1b[0m' // Reset
  };

  let verboseLogging = false;

  export function setVerboseLogging(span class="hljs-params">verbose: boolean/span>):
   void {                                  !a
    verboseLogging = verbose;
  }

  export function log(span class="hljs-params">message: string, level: LogLevel =
   LogLevel.INFO/span>): void {           !a
    // Only log DEBUG messages if verbose logging is enabled
    if (level === LogLevel.DEBUG && !verboseLogging) {
      return;
    }

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

    console.log(span class="hljs-string">`${color}[${timestamp}] [${level}] ${message}
   ${COLORS.RESET}`/span>););             !a
  }
```

syntax/highlighter.ts Language: TYPESCRIPT

```typescript
import hljs from 'highlight.js';
import { FileInfo } from '../file-reader';
import { log, LogLevel } from '../utils/logger';

// Language mapping for extensions not automatically recognized by highlight.js
const LANGUAGE_MAP: Recordstring, string> = {
  'ts': 'typescript',
  'js': 'javascript',
  'jsx': 'javascript',
  'tsx': 'typescript',
  'md': 'markdown',
  'yml': 'yaml',
  // Add more mappings as needed
};

export interface HighlightedLine {
  line: string;
  lineNumber: number;
  tokens: {
    text: string;
    color?: string;
    fontStyle?: string;
    appendSpace?: boolean;  // New property to indicate if space should be appended
  }[];
}

export interface HighlightedFile extends FileInfo {
  highlightedLines: HighlightedLine[];
  language: string;
}

// Function to get language for syntax highlighting
function getLanguage(span class="hljs-params">extension: string/span>): string {
  return LANGUAGE_MAP[extension.toLowerCase()] || extension.toLowerCase();
}

// Improved HTML entity decoder
function decodeHtmlEntities(span class="hljs-params">text: string/span>): string {
  // Handle common HTML entities
  const entityMap: Recordstring, string> = {
    '&amp;': '&',
span class="hljs-string">'    '/span>: span class="hljs-string">''/span>,: '<',
'>': '>',
'"': '"',
''': "'",
```

syntax/highlighter.ts

Language: TYPESCRIPT

```
  ''': "'",
  '/': '/',
  '`': '`',
    };

    // First pass: replace named entities
    let result = text;
    for (const [entity, replacement] of Object.entries(entityMap)) {
      result = result.replace(new RegExp(entity, 'g'), replacement);
    }

    // Second pass: handle numeric entities
    result = result.replace(/&#(\d+);/g, (_, dec) => String.fromCharCode(parseInt(dec, 10)));
    result = result.replace(/&#x([0-9a-f]+);/gi, (_, hex) => String.fromCharCode(parseInt(hex, 16)));

    return result;
  }

  // Main function to highlight code
  export function highlightCode(fileInfo: FileInfo): HighlightedFile {
    const language = getLanguage(fileInfo.extension);

    try {
      // Split content into lines
      const lines = fileInfo.content.split('\n');

      const highlightedLines: HighlightedLine[] = lines.map((line, index) => {
        // Skip highlighting if line is empty
        if (line.trim() === '') {
          return {
            line,
            lineNumber: index + 1,
            tokens: [{ text: '' }]
          };
        }

        let highlighted;

        // Try to highlight with specific language
        try {
          highlighted = hljs.highlight(line, { language });
```

```
        } catch (e) {
          // Fall back to auto detection
          highlighted = hljs.highlightAuto(line);
        }

        // Extract tokens using the improved approach with space preservation
        const tokens = extractTokensWithSpaces(line, highlighted.value);

        return {
          line,
          lineNumber: index + 1,
          tokens
        };
      });

      return {
        ...fileInfo,
        highlightedLines,
        language
      };
    } catch (err) {
      log(span class="hljs-string">`Error highlighting code for ${fileInfo.path}:
     span class="hljs-subst">${(err as Error).message}/span>`/span>, LogLevel.ERROR);

      // Return basic line-by-line structure without highlighting
      const lines = fileInfo.content.split('\n');
      const highlightedLines = lines.map(span class="hljs-function">(line, index) =>
     /span> ({ ({
        line,
        lineNumber: index + 1,
        tokens: [{ text: line }]
      }));

      return {
        ...fileInfo,
        highlightedLines,
        language
      };
    }
  }

  // Advanced token extraction that preserves spaces between keywords
  function extractTokensWithSpaces(span class="hljs-params">originalLine: string, html:
     string/span>): { text: string; color?: string; fontStyle?: string; appendSpace?:
     boolean }[] {
```

syntax/highlighter.ts Language: TYPESCRIPT

```
    const tokens: { text: string; color?: string; fontStyle?: string; appendSpace?:
     boolean }[] = [];                    !a

    // Step 1: Clean up the HTML and decode entities
    const cleanedHtml = decodeHtmlEntities(html);

    // Step 2: Extract tokens with a more robust approach
    // This regex matches span elements or text nodes
    const tokenRegex = span class="hljs-regexp">/span class="([^"]+)">([^]+)\/span>|([^
     ]+)/g/span>;;                        !a
    let match;

    // Keep track of current position in the original line for space detection
    let currentPos = 0;

    while ((match = tokenRegex.exec(cleanedHtml)) !== null) {
      let tokenText = '';
      let className = '';

      if (match[3]) {
        // Plain text (not in span)
        tokenText = match[3];
      } else {
        // Text with highlighting in a span
        className = match[1]; // Class like "hljs-keyword", etc.
        tokenText = match[2];
      }

      if (tokenText.trim()) {
        const color = getColorForClass(className);
        const fontStyle = getFontStyleForClass(className);

        // Find this token's position in the original line
        const tokenPos = originalLine.indexOf(tokenText, currentPos);
        if (tokenPos !== -1) {
          // Check if there are spaces before this token that need to be preserved
          if (tokenPos > currentPos) {
            const spaces = originalLine.substring(currentPos, tokenPos);
            if (spaces.trim() === '') {
              // Add spaces as a separate token
              tokens.push({ text: spaces });
            }
          }

          // Add the actual token
```

syntax/highlighter.ts Language: TYPESCRIPT

```
171           tokens.push({ text: tokenText, color, fontStyle });
172
173           // Update current position for next token
174           currentPos = tokenPos + tokenText.length;
175         } else {
176           // Fallback if we can't find the exact position
177           tokens.push({ text: tokenText, color, fontStyle });
178         }
179       }
180     }
181
182     // Check if there are any remaining spaces at the end
183     if (currentPos  originalLine.length) {
184       const remainingText = originalLine.substring(currentPos);
185       if (remainingText.trim() !== '') {
186         tokens.push({ text: remainingText });
187       }
188     }
189
190     return tokens;
191   }
192
193   // Improved color mapping for syntax highlighting
194   function getColorForClass(span class="hljs-params">className: string/span>): string |
       undefined {                                  !a
195     if (className.includes('keyword')) return '#0000ff'; // Blue
196     if (className.includes('string')) return '#008000'; // Green
197     if (className.includes('comment')) return '#808080'; // Gray
198     if (className.includes('number')) return '#009999'; // Teal
199     if (className.includes('function')) return '#AA6E28'; // Brown
200     if (className.includes('title')) return '#900'; // Dark Red
201     if (className.includes('params')) return '#444'; // Dark Gray
202     if (className.includes('built_in')) return '#0086b3'; // Light Blue
203     if (className.includes('literal')) return '#990073'; // Purple
204     if (className.includes('property')) return '#905'; // Pink
205     if (className.includes('operator')) return '#9a6e3a'; // Dark Brown
206     if (className.includes('punctuation')) return '#333'; // Dark Gray
207     if (className.includes('attr')) return '#0086b3'; // Light Blue (for attributes)
208     return undefined;
209   }
210
211   // Map classes to font styles
212   function getFontStyleForClass(span class="hljs-params">className: string/span>):
       string | undefined {                         !a
213     if (className.includes('comment')) return 'italic';
```

syntax/highlighter.ts Language: TYPESCRIPT

```typescript
    if (className.includes('bold')) return 'bold';
    if (className.includes('italic')) return 'italic';
    if (className.includes('emphasis')) return 'italic';
    if (className.includes('strong')) return 'bold';
    return undefined;
  }
```

pdf/toc.ts Language: TYPESCRIPT

```typescript
import PDFDocument from 'pdfkit';
import { HighlightedFile } from '../syntax/highlighter';
import { PdfOptions } from './generator';

export function generateTOC(span class="hljs-params">/span>
  doc: PDFKit.PDFDocument,
  files: HighlightedFile[],
  options: PdfOptions
): void {
  // Add a new page for TOC
  doc.addPage();

  // Set up page dimensions
  const pageWidth = options.paperSize![0] - options.margins!.left - options.margins!.
   right;                                  !a

  // Add TOC title
  doc.font('Helvetica-Bold')
     .fontSize(18)
     .text('Table of Contents', { align: 'center' })
     .moveDown(2);

  // Group files by directories for a hierarchical TOC
  const filesByDirectory: Recordstring, HighlightedFile[]> = {};

  files.forEach(span class="hljs-function">file =>/span> { {
    const dirPath = file.relativePath.split('/').slice(0, -1).join('/');
    if (!filesByDirectory[dirPath]) {
      filesByDirectory[dirPath] = [];
    }
    filesByDirectory[dirPath].push(file);
  });

  // Calculate actual page numbers
  // Cover page + TOC page = 2 pages before files
  let currentPage = 3;
  const pageNumbers: Recordstring, number> = {};

  // Calculate page numbers first
  const directories = Object.keys(filesByDirectory).sort();
  directories.forEach(span class="hljs-function">directory =>/span> { {
    const sortedFiles = filesByDirectory[directory].sort(span class="hljs-function">(
   a, b) =>/span>                          !a
      a.relativePath.localeCompare(b.relativePath)
    );
```

```
44
45        sortedFiles.forEach(span class="hljs-function">file =>/span> { {
46          pageNumbers[file.relativePath] = currentPage;
47
48          // Calculate realistic page count based on file size
49          const lineCount = file.highlightedLines.length;
50          const linesPerPage = Math.floor((options.paperSize![1] - options.margins!.top
        - options.margins!.bottom -          !a
51                                  options.headerHeight! - options.footerHeight!) / (options.
      fontSize * 1.4));                      !a
52          const estimatedPages = Math.max(1, Math.ceil(lineCount / linesPerPage));
53          currentPage += estimatedPages;
54        });
55      });
56
57      // Now render the TOC with accurate page numbers
58      directories.forEach(span class="hljs-function">directory =>/span> { {
59        if (directory) {
60          // Add directory name with better formatting
61          doc.font('Helvetica-Bold')
62             .fontSize(14)
63             .fillColor('#000000');
64
65          // Add a box around the directory name
66          const directoryText = directory;
67          const textWidth = doc.widthOfString(directoryText);
68          const textHeight = doc.currentLineHeight();
69
70          doc.rect(
71            options.margins!.left,
72            doc.y,
73            pageWidth,
74            textHeight + 8
75          )
76          .fillColor('#f0f0f0')
77          .fill();
78
79          // Write directory name
80          doc.fillColor('#000000')
81             .text(directoryText, options.margins!.left + 10, doc.y - textHeight + 4);
82
83          doc.moveDown(1);
84        }
85
86        // Sort files in the directory
```

pdf/toc.ts Language: TYPESCRIPT

```typescript
    const sortedFiles = filesByDirectory[directory].sort(span class="hljs-function">(a, b) =>/span>
      a.relativePath.localeCompare(b.relativePath)
    );

    // Add file entries
    sortedFiles.forEach(span class="hljs-function">file =>/span> { {
      const fileName = file.relativePath.split('/').pop() || file.relativePath;
      const indent = directory ? '    ' : '';

      // Get page number for this file
      const pageNum = pageNumbers[file.relativePath];

      // Calculate positions
      const startX = options.margins!.left + (directory ? 20 : 0);
      const pageNumWidth = doc.widthOfString(String(pageNum));
      const endX = options.margins!.left + pageWidth - pageNumWidth;
      const nameWidth = endX - startX - 20; // Leave space for dots

      // Add file name
      doc.font('Helvetica')
         .fontSize(12)
         .fillColor('#000000')
         .text(span class="hljs-string">`${indent}${fileName}`/span>, startX, doc.y, {
           continued: true,
           width: nameWidth
         });

      // Create a dot leader that's more compact
      const dotLeader = '. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ';
      doc.fillColor('#888888')
         .text(dotLeader, { continued: true });

      // Add page number
      doc.fillColor('#000000')
         .font('Helvetica-Bold')
         .text(span class="hljs-string">` ${pageNum}`/span>, { align: 'right' });

      doc.moveDown(0.5);
    });

    doc.moveDown(0.5);
  });
```

pdf/toc.ts Language: TYPESCRIPT

```typescript

    // Add note about page numbers
    doc.moveDown(2)
       .font('Helvetica-Oblique')
       .fontSize(10)
       .fillColor('#555555')
       .text('Note: Page numbers reflect actual document pagination.', {
         align: 'center',
         width: pageWidth
       });
  }
```

pdf/page.ts Language: TYPESCRIPT

```typescript
import PDFDocument from 'pdfkit';
import { HighlightedFile } from '../syntax/highlighter';
import { PdfOptions } from './generator';

// Track page number across page renders - global counter
let currentPageNumber = 1;

// Define token type for better type safety
interface RenderToken {
  text: string;
  color?: string;
  fontStyle?: string;
}

export function renderPage(span class="hljs-params">/span>
  doc: PDFKit.PDFDocument,
  file: HighlightedFile,
  options: PdfOptions
): void {
  const pageWidth = options.paperSize![0] - options.margins!.left - options.margins!.
   right;
  const contentHeight = options.paperSize![1] - options.margins!.top - options.margins
   !.bottom - options.headerHeight! - options.footerHeight!;

  // Add header with file path
  renderHeader(doc, file, options);

  // Calculate starting position after header
  const startY = options.margins!.top + options.headerHeight!;
  doc.y = startY;

  // Calculate line number column width based on the number of lines
  const maxLineNumber = file.highlightedLines.length;
  const lineNumberWidth = options.showLineNumbers ? Math.max(String(maxLineNumber).
   length * options.fontSize * 0.8, 50) : 0;

  // Render code
  renderCodeBlockSimple(doc, file, options, lineNumberWidth, startY, contentHeight);

  // Add footer with page number
  renderFooter(doc, options, currentPageNumber);

  // Increment page counter after rendering the page
  currentPageNumber++;
}
```

pdf/page.ts Language: TYPESCRIPT

```typescript

function renderHeader(span class="hljs-params">/span>
  doc: PDFKit.PDFDocument,
  file: HighlightedFile,
  options: PdfOptions
): void {
  const headerY = options.margins!.top;
  const pageWidth = options.paperSize![0] - options.margins!.left - options.margins!.
   right;

  // Draw background for header
  doc.rect(options.margins!.left, headerY, pageWidth, options.headerHeight!)
     .fillColor('#f8f8f8')
     .fill();

  // Draw file path
  doc.font('Helvetica-Bold')
     .fontSize(12)
     .fillColor('#333333')
     .text(file.relativePath, options.margins!.left + 10, headerY + 8, {
       width: pageWidth - 150,
       align: 'left'
     });

  // Draw language
  doc.font('Helvetica')
     .fontSize(10)
     .fillColor('#666666')
     .text(span class="hljs-string">`Language: ${file.language.toUpperCase()}`
   /span>, options.margins!.left + pageWidth - 140, headerY + 8, {
       width: 130,
       align: 'right'
     });

  // Draw a line under the header
  doc.moveTo(options.margins!.left, headerY + options.headerHeight! - 1)
     .lineTo(options.paperSize![0] - options.margins!.right, headerY + options.
   headerHeight! - 1)
     .lineWidth(1)
     .strokeColor('#dddddd')
     .stroke();
}

function renderFooter(span class="hljs-params">/span>
  doc: PDFKit.PDFDocument,
```

pdf/page.ts Language: TYPESCRIPT

```
    options: PdfOptions,
    pageNumber: number
  ): void {
    const footerY = options.paperSize![1] - options.margins!.bottom - options.
      footerHeight!;                                    !a
    const pageWidth = options.paperSize![0] - options.margins!.left - options.margins!.
      right;                                            !a

    // Draw a line above the footer
    doc.moveTo(options.margins!.left, footerY)
       .lineTo(options.paperSize![0] - options.margins!.right, footerY)
       .lineWidth(1)
       .strokeColor('#dddddd')
       .stroke();

    // Draw page number
    doc.font('Helvetica')
       .fontSize(10)
       .fillColor('#666666')
       .text(span class="hljs-string">`Page ${pageNumber}`/span>, options.margins!.left
      , footerY + 10, {                                 !a
         width: pageWidth,
         align: 'center'
       });
  }

  function renderCodeBlockSimple(span class="hljs-params">/span>
    doc: PDFKit.PDFDocument,
    file: HighlightedFile,
    options: PdfOptions,
    lineNumberWidth: number,
    startY: number,
    contentHeight: number
  ): void {
    // Set consistent monospace font
    doc.font('Courier')
       .fontSize(options.fontSize);

    // Calculate available width for code
    const codeWidth = options.paperSize![0] - options.margins!.left - options.margins!.
      right - lineNumberWidth - 30;                     !a

    // Calculate line height (increased for better readability)
    const lineHeight = options.fontSize * 1.8;
```

pdf/page.ts

Language: TYPESCRIPT

```typescript
    // Current position for drawing
    let currentY = startY + 10;

    // Draw background for the code block
    doc.rect(options.margins!.left, startY, options.paperSize![0] - options.margins!.left - options.margins!.right, contentHeight)
       .fill('#f8f8f8');

    // Draw line number background if line numbers are shown
    if (options.showLineNumbers) {
      doc.rect(options.margins!.left, startY, lineNumberWidth, contentHeight)
         .fill('#e8e8e8');
    }

    // Process each line of code
    for (let i = 0; i  file.highlightedLines.length; i++) {
      const line = file.highlightedLines[i];

      // Starting X position for code content
      const codeX = options.margins!.left + (options.showLineNumbers
       ? lineNumberWidth + 15 : 15);

      // Check if we need a new page
      if (currentY + lineHeight > startY + contentHeight) {
        // Add footer to current page
        renderFooter(doc, options, currentPageNumber);

        // Add a new page
        doc.addPage();

        // Increment page counter
        currentPageNumber++;

        // Reset current Y position
        currentY = startY + 10;

        // Add header to new page
        renderHeader(doc, file, options);

        // Draw background for the code block on the new page
        doc.rect(options.margins!.left, startY, options.paperSize![0] - options.margins!.left - options.margins!.right, contentHeight)
           .fill('#f8f8f8');

        // Draw line number background if line numbers are shown
```

pdf/page.ts

Language: TYPESCRIPT

```
        if (options.showLineNumbers) {
          doc.rect(options.margins!.left, startY, lineNumberWidth, contentHeight)
             .fill('#e8e8e8');
        }
      }

      // Draw line number if enabled
      if (options.showLineNumbers) {
        doc.font('Courier-Bold')
           .fontSize(options.fontSize)
           .fillColor('#888888')
           .text(
             String(line.lineNumber).padStart(String(file.highlightedLines.length).
    length, ' '),                          !a
             options.margins!.left + 5,
             currentY,
             { width: lineNumberWidth - 10, align: 'right' }
           );
      }

      // Calculate total width of the line (including spaces)
      let totalWidth = 0;
      for (const token of line.tokens) {
        doc.font(token.fontStyle === 'bold' ? 'Courier-Bold' :
                 token.fontStyle === 'italic' ? 'Courier-Oblique' : 'Courier');
        totalWidth += doc.widthOfString(token.text);
      }

      // Check if line needs wrapping
      if (totalWidth = codeWidth) {
        // Simple case: Draw each token sequentially
        let xPos = codeX;
        for (const token of line.tokens) {
          doc.font(token.fontStyle === 'bold' ? 'Courier-Bold' :
                   token.fontStyle === 'italic' ? 'Courier-Oblique' : 'Courier')
             .fillColor(token.color || '#000000');

          doc.text(token.text, xPos, currentY, { continued: false });
          xPos += doc.widthOfString(token.text);
        }

        currentY += lineHeight;

      } else {
        // Handle wrapped lines
```

pdf/page.ts Language: TYPESCRIPT

```
        // Two-pass approach: First break into lines, then render
        const virtualLines: RenderToken[][] = [];
        let currentVirtualLine: RenderToken[] = [];
        let currentLineWidth = 0;

        // First pass: Determine line breaks
        for (const token of line.tokens) {
          const font = token.fontStyle === 'bold' ? 'Courier-Bold' :
                       token.fontStyle === 'italic' ? 'Courier-Oblique' : 'Courier';
          doc.font(font);

          // If token would make line too long, create a new virtual line
          if (currentLineWidth + doc.widthOfString(token.text) > codeWidth) {
            // If token itself is very long, we need to split it
            if (doc.widthOfString(token.text) > codeWidth / 2) {
              // Split long token into parts
              let remainingText = token.text;

              while (remainingText.length > 0) {
                // Find maximum characters that can fit
                let charsThatFit = 0;
                let spaceLeft = codeWidth - currentLineWidth;

                if (spaceLeft  doc.widthOfString('W')) {
                  // Not enough space on current line, add to next line
                  virtualLines.push([...currentVirtualLine]);
                  currentVirtualLine = [];
                  currentLineWidth = 0;
                  spaceLeft = codeWidth;
                }

                // Try to fit as many characters as possible
                for (let j = 1; j = remainingText.length; j++) {
                  const partWidth = doc.widthOfString(remainingText.substring(0, j));
                  if (partWidth = spaceLeft) {
                    charsThatFit = j;
                  } else {
                    break;
                  }
                }

                if (charsThatFit > 0) {
                  const partText = remainingText.substring(0, charsThatFit);
                  const partToken: RenderToken = {
                    text: partText,
```

```
                    color: token.color,
                    fontStyle: token.fontStyle
                  };

                  currentVirtualLine.push(partToken);
                  currentLineWidth += doc.widthOfString(partText);
                  remainingText = remainingText.substring(charsThatFit);
                }

                if (remainingText.length > 0) {
                  // We have more text that needs to go to the next line
                  virtualLines.push([...currentVirtualLine]);
                  currentVirtualLine = [];
                  currentLineWidth = 0;
                }
              }
            } else {
              // Token doesn't fit on current line but isn't too long
              if (currentVirtualLine.length > 0) {
                virtualLines.push([...currentVirtualLine]);
              }
              currentVirtualLine = [token];
              currentLineWidth = doc.widthOfString(token.text);
            }
          } else {
            // Token fits on current line
            currentVirtualLine.push(token);
            currentLineWidth += doc.widthOfString(token.text);
          }
        }

        // Add the last virtual line if it has content
        if (currentVirtualLine.length > 0) {
          virtualLines.push(currentVirtualLine);
        }

        // Second pass: Render each virtual line
        let isFirstLine = true;
        for (const vLine of virtualLines) {
          // Check if we need a new page
          if (currentY + lineHeight > startY + contentHeight) {
            renderFooter(doc, options, currentPageNumber);
            doc.addPage();
            currentPageNumber++;
            currentY = startY + 10;
```

```typescript
            renderHeader(doc, file, options);

            // Redraw backgrounds
            doc.rect(options.margins!.left, startY, options.paperSize![0] - options.margins!.left - options.margins!.right, contentHeight)
              .fill('#f8f8f8');

            if (options.showLineNumbers) {
              doc.rect(options.margins!.left, startY, lineNumberWidth, contentHeight)
                .fill('#e8e8e8');
            }
          }

          // For wrapped lines after the first, show continuation marker
          if (!isFirstLine && options.showLineNumbers) {
            doc.font('Courier')
               .fontSize(options.fontSize)
               .fillColor('#888888')
               .text('↪', options.margins!.left + lineNumberWidth/2 - 10, currentY, { align: 'center' });
          }

          // Draw tokens for this virtual line
          let xPos = codeX;
          // Add indentation for continuation lines
          if (!isFirstLine) {
            xPos += options.fontSize * 2;
          }

          for (const token of vLine) {
            doc.font(token.fontStyle === 'bold' ? 'Courier-Bold' :
                     token.fontStyle === 'italic' ? 'Courier-Oblique' : 'Courier')
               .fillColor(token.color || '#000000');

            doc.text(token.text, xPos, currentY, { continued: false });
            xPos += doc.widthOfString(token.text);
          }

          currentY += lineHeight;
          isFirstLine = false;
        }
      }
    }
  }
```

pdf/generator.ts Language: TYPESCRIPT

```typescript
import PDFDocument from 'pdfkit';
import fs from 'fs-extra';
import path from 'path';
import { readPath } from '../file-reader';
import { highlightCode, HighlightedFile } from '../syntax/highlighter';
import { renderPage } from './page'; // Removed resetPageCounter import
import { generateTOC } from './toc';
import { log, LogLevel } from '../utils/logger';

export interface PdfOptions {
  title: string;
  theme: string;
  fontSize: number;
  showLineNumbers: boolean;
  paperSize?: [number, number]; // Width, height in points (default A4)
  margins?: { top: number; right: number; bottom: number; left: number };
  headerHeight?: number;
  footerHeight?: number;
}

// Default options - with better default sizes
const defaultOptions: PdfOptions = {
  title: 'Code Documentation',
  theme: 'github',
  fontSize: 11, // Increased from 10
  showLineNumbers: true,
  paperSize: [595.28, 841.89], // A4
  margins: { top: 50, right: 50, bottom: 50, left: 50 },
  headerHeight: 40, // Increased slightly
  footerHeight: 30
};

export async function generatePdfFromPath(span class="hljs-params">/span>
  inputPath: string,
  outputPath: string,
  options: PartialPdfOptions> = {}
): Promisevoid> {
  // Merge options with defaults
  const mergedOptions: PdfOptions = { ...defaultOptions, ...options };

  try {
    // Read all files
    log(span class="hljs-string">`Reading files from ${inputPath}...`/span>, LogLevel.
    INFO);
    const files = await readPath(inputPath);
```

pdf/generator.ts Language: TYPESCRIPT

```typescript

      // Highlight code for all files
      log('Applying syntax highlighting...', LogLevel.INFO);
      const highlightedFiles: HighlightedFile[] = files.map(span class="hljs-function">
      file =>/span>  highlightCode(file));
      // Create PDF document
      log('Generating PDF...', LogLevel.INFO);
      const doc = new PDFDocument({
        size: mergedOptions.paperSize,
        margins: mergedOptions.margins,
        info: {
          Title: mergedOptions.title,
          Author: 'Generated by xprinto',
          Creator: 'xprinto'
        }
      });

      // Create write stream
      const writeStream = fs.createWriteStream(outputPath);
      doc.pipe(writeStream);

      // Add cover page
      addCoverPage(doc, inputPath, mergedOptions);

      // Add table of contents if there are multiple files
      if (highlightedFiles.length > 1) {
        log('Generating table of contents...', LogLevel.INFO);
        generateTOC(doc, highlightedFiles, mergedOptions);
      }

      // Add each file to the PDF
      for (const file of highlightedFiles) {
        log(span class="hljs-string">`Adding file to PDF: ${file.relativePath}`/span>,
      LogLevel.INFO);
        doc.addPage();
        renderPage(doc, file, mergedOptions);
      }

      // Finalize the PDF
      doc.end();

      // Wait for the write stream to finish
      await new Promisevoid>(span class="hljs-function">(resolve, reject) =>/span> { {
        writeStream.on('finish', () => {
```

pdf/generator.ts Language: TYPESCRIPT

```
88            log(span class="hljs-string">`PDF written to ${outputPath}`/span>, LogLevel.
      SUCCESS);                                  !a
89            resolve();
90          });
91          writeStream.on('error', reject);
92        });
93      } catch (err) {
94        throw new Error(span class="hljs-string">`Failed to generate PDF:
        span class="hljs-subst">${(err as Error).message}/span>`/span>);`);
95      }
96    }
97
98    // Function to add a cover page
99    function addCoverPage(span class="hljs-params">doc: PDFKit.PDFDocument, inputPath:
        string, options: PdfOptions/span>): void {
100     // Set font for cover page
101     doc.font('Helvetica-Bold')
102        .fontSize(24)
103        .text(options.title, { align: 'center' })
104        .moveDown(2);
105
106     // Add input path information
107     doc.font('Helvetica')
108        .fontSize(14)
109        .text(span class="hljs-string">`Source: ${path.resolve(inputPath)}`/span>, {
        align: 'center' })                       !a
110        .moveDown(1);
111
112     // Add generation date
113     doc.fontSize(12)
114        .text(span class="hljs-string">`Generated on: span class="hljs-subst">${new Date
        ().toLocaleString()}/span>`/span>, { align: 'center' })
115        .moveDown(4);
116
117     // Add separator line
118     const pageWidth = options.paperSize![0] - options.margins!.left - options.margins!.
        right;                                   !a
119     doc.moveTo(options.margins!.left, doc.y)
120        .lineTo(options.margins!.left + pageWidth, doc.y)
121        .stroke();
122
123     // Add description
124     doc.moveDown(2)
125        .fontSize(12)
126        .text('This document contains a formatted representation of the code with syntax
```

pdf/generator.ts Language: TYPESCRIPT

```
    highlighting and line numbers for easy reference. Navigate through the document using
    the table of contents (if available).", {
        align: 'left',
        width: pageWidth
      });
}
```