

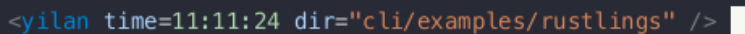
Behind the Scenes of CodeSlide CLI

```
<code id="slide" class="cli">  
  CodeSlide CLI  
</code>
```

CodeSlide CLI

npm v0.12.13

Usage demo



```
<yil time=11:11:24 dir="cli/examples/rustlings" />
```

See also [Example usages](#)

Installation

1. Prepare Node.js runtime and NPM package manager
2. Run `npm install -g codeslide-cli` on the command line

Features

- It allows you to easily make awesome slideshows for code snippets on command lines
- It is an application of [CodeSlide](#)
- It is a Node.js Command Line Interface (CLI)

Documents

- See [Reference](#) for more information

Creator

- [AsherJingkongChen](#)

The general process

1. Build a schema
2. Render HTML and CSS to slideshow with it
3. Print the rendered slideshow to the output

Manifest

- The main schema
- A combination of [FrontMatter](#) and [SlideShow](#)
- An extended [Renderex](#) schema
- `Manifest.parse`: Parse a [manifest file](#) into a Manifest schema
- `Manifest.render`: Render the slideshow

```
import matter from 'gray-matter';
import { launch } from 'puppeteer';
import { FrontMatter } from './FrontMatter';
import { SlideShow } from './SlideShow';
```

```

import { Renderer } from '../../../src';

export type Manifest = FrontMatter & SlideShow;

export namespace Manifest {
  export type Result = {
    data: string,
    encoding: BufferEncoding,
  };

  export const parse = async (
    manifest: string
  ): Promise<Manifest> => {
    manifest = manifest.replace(
      /\[\u200B\u200C\u200D\u200E\u200F\uFEFF]/, ''
    );
    const { content, data: { codeslide } } = matter(manifest);
    const fm = await FrontMatter.parse(codeslide);
    const slides = await SlideShow.parse(content);
    return { ...fm, ...slides };
  };

  export const render = async (
    manifest: Manifest
  ): Promise<Result> => {
    if (manifest.format === 'html') {
      return {
        data: Renderer.render(manifest),
        encoding: 'utf8',
      };
    } else {
      const browser = await launch();
      const page = await browser.newPage();
      await page.setContent(Renderer.render(manifest));
      const result = await page.pdf({
        printBackground: true,
        format: manifest.pageSize,
      });
      await browser.close();
      return {
        data: result.toString('base64'),
        encoding: 'base64',
      };
    }
  };
}

```

Manifest file

- A markdown document constructed of the Front Matter section and the Slide Show section
 - The specifications of Manifest file is [here](#)
-

FrontMatter

- The Front Matter section schema
- An extended `Renderer` schema
- Parsed from the Front Matter section of [manifest file](#) (YAML syntax)

```
import semver from 'semver-regex';
import { z } from 'zod';
import { Renderer } from '../../../../../src';
import { version } from '../../../../../package.json';
import { formatZodError, getContent } from '../utils';

export type FrontMatter = z.infer<typeof FrontMatter.schema>;

export namespace FrontMatter {
  export const parse = async (
    fm?: Partial<FrontMatter>
  ): Promise<FrontMatter> => (
    schema.default({}).parseAsync(fm)
  );

  export const schema = z
    .object({
      format: z.enum(['html', 'pdf']).default('html'),
      pageSize: z
        .enum([
          'letter', 'legal', 'tabloid', 'ledger',
          'A0', 'A1', 'A2', 'A3', 'A4', 'A5', 'A6'
        ])
        .default('A4'),
    })
```

```

    version: z.string().regex(semver()).default(version),
  })
  .and(Renderer.schema.omit({ slides: true }))
  .transform(async (fm) => {
    fm.styles = await Promise.all(
      fm.styles.map((path) => getContent(path))
    );
    if (fm.codeFont.rule) {
      fm.codeFont.rule = await getContent(fm.codeFont.rule);
    }
    if (fm.slideFont.rule) {
      fm.slideFont.rule = await getContent(fm.slideFont.rule);
    }
    return fm;
  })
  .catch((e) => {
    const error = e.error.errors[0];
    error.path.unshift('codeslide');
    throw new Error(`\
Cannot parse the Front Matter section:
\t${formatZodError(error)}\
`);
  })
  .transform((fm) => {
    if (
      fm.format === 'pdf' &&
      fm.layout !== 'vertical'
    ) {
      fm.layout = 'vertical';
    }
    return fm;
  });
}

```

SlideShow

- The Slide Show section schema
- An labeled object whose type is { slides: string[] }
- Parsed from the Slide Show section of [manifest file](#) (Markdown syntax)
- Each slide is splitted by a horizontal line

- Has special rules for rendering embedding code snippets and slides

```
import hljs, { HighlightResult } from 'highlight.js';
import { marked } from 'marked';
import { getContent } from '../utils';

export type SlideShow = { slides: string[] };

export namespace SlideShow {
  export const parse = async (
    markdown: string
  ): Promise<SlideShow> => {
    const html = await _parseMarkdown(markdown);
    return { slides: html.split('<hr>').map((s) => s.trim()) };
  };
}

const _parseMarkdown = (
  markdown: string
): Promise<string> => (
  marked.parse(markdown, {
    async: true,
    highlight,
    renderer,
    walkTokens,
  }).catch((err: Error) => {
    err.message = `Cannot parse the Slide Show section:\n\t${
      err.message.replace(
        '\nPlease report this to https://github.com/markedjs/marked.',
        ''
      )
    }`;
    throw err;
  })
);

const highlight = (code: string, language: string): string => {
  try {
    return hljs.highlight(code, { language }).value;
  } catch (e) {
    const err = e as Error;
    err.message =
      `Cannot parse the code "${
        code.substring(0, 30).split('\n')[0]
      } ...":\n\t${err.message}`;
    throw e;
  }
}
```

```

    }
};

const renderer = new class extends marked.Renderer {
  override code(
    code: string,
    language: string | undefined,
    isEscaped: boolean
  ): string {
    const old = super.code(code, language, isEscaped);
    const index = old.indexOf('>');
    if (index === -1) {
      return old;
    }
    return `${old.slice(0, index)} hljs${old.slice(index)}`;
  }
};

```

```

const walkTokens = async (
  token: marked.Token
): Promise<void> => {
  if (token.type === 'link') {
    const { href, text } = token;
    if (!text.startsWith(':')) {
      return;
    }
    const [prefix, suffix] = <[string, string | undefined]>
      text.split('.');
    token = _toHTMLToken(token);
    if (prefix === ':slide') {
      const slide = await getContent(href);
      token.text = await _parseMarkdown(slide);
    } else if (prefix === ':code') {
      const code = await getContent(href);
      let result: HighlightResult | undefined;
      try {
        result = hljs.highlight(code, {
          language: suffix ?? 'plaintext'
        });
      } catch (e) {
        const err = e as Error;
        err.message = `
Cannot parse the code at ${href}:
\t${err.message}`;
        throw e;
      }
    }
  }
}

```

```

        token.text = `\  

<pre><code${  

    result.language ?  

        ` class="language-${result.language} hljs"` : ''  

}>${  

    result.value  

}</code></pre>`;
    }  

}
};

const _toHTMLToken = (  

    token: marked.Token  

): marked.Tokens.HTML => {  

    const { raw } = token;  

    for (const p in token) {  

        if (token.hasOwnProperty(p)){  

            delete token[p as keyof marked.Token];  

        }  

    }  

    token = token as marked.Token;  

    token.type = 'html';  

    token.raw = raw;  

    token = token as marked.Tokens.HTML;  

    token.pre = true;  

    return token;  

};

```

Renderer

- The slideshow renderer schema
- Depends on [Eta](#) to render [HTML template](#)
- `Renderer.parse`: Parse an object into a Renderer schema
- `Renderer.render`: Render the slideshow to HTML text

Note

- `Renderer` is the root schema of [Manifest](#). That is, [Manifest](#) is an extended `Renderer`.


```

declare module '*.css' {
  const _: string;
  export default _;
}
declare module '*.html' {
  const _: string;
  export default _;
}

import VisualStudio from './VisualStudio.css';
import HorizontalStylesheet from './slides.horizontal.css';
import VerticalStylesheet from './slides.vertical.css';
import HTMLTemplate from './slides.html';

const Stylesheets = {
  horizontal: HorizontalStylesheet,
  vertical: VerticalStylesheet,
  highlight: VisualStudio,
};

export { HTMLTemplate, Stylesheets };

import { render as renderEta } from 'eta';
import { z } from 'zod';
import { Stylesheets, HTMLTemplate } from '../assets';

export type Renderer = z.infer<typeof Renderer.schema>;

export namespace Renderer {
  export const parse = (
    renderer?: Partial<Renderer>
  ): Renderer => (
    schema.default({}).parse(renderer)
  );

  export const render = (
    renderer: Renderer
  ): string => {
    let {
      codeFont, slideFont,
      layout, slides, styles,
    } = renderer;

    const preStyles = new Array<string>();

```

```

    if (! styles.length) {
      preStyles.push(Stylesheets['highlight']);
    }
    preStyles.push(Stylesheets[layout]);
    styles.unshift(...preStyles);

    if (codeFont.rule) {
      styles.push(
        '/*! CodeSlide codeFont.rule */',
        codeFont.rule,
      );
    }
    if (slideFont.rule) {
      styles.push(
        '/*! CodeSlide slideFont.rule */',
        slideFont.rule,
      );
    }
    styles.push(`\
/* CodeSlide codeFont properties */
code {
  font-family: ${codeFont.family};
}
pre > code {
  font-size: ${codeFont.size};
  font-weight: ${codeFont.weight};
}

/* CodeSlide slideFont properties */
#slides {
  font-family: ${slideFont.family};
  font-size: ${slideFont.size};
  font-weight: ${slideFont.weight};
}`);
    const style = `<style>${styles.join('\n')}</style>`;

    return renderEta(
      HTMLTemplate,
      {
        layout,
        slides,
        style,
      },
      {
        autoTrim: false,
        tags: ['{%', '%}']
      }
    );
  }
}

```

```

    }
  );
};

export const schema = z
  .object({
    codeFont: z
      .object({
        family: z.string().optional()
          .transform((arg) => `
${arg ? `${arg}`, ` : ''}ui-monospace, SFMono-Regular, \
SF Mono, Menlo, Consolas, Liberation Mono, monospace`
          ),
        rule: z.string().optional(),
        size: z.string().default('medium'),
        weight: z.string().default('normal'),
      })
      .default({}),
    slideFont: z
      .object({
        family: z.string().optional()
          .transform((arg) => `${arg ? `${arg}`, ` : ''}system-ui`),
        rule: z.string().optional(),
        size: z.string().default('large'),
        weight: z.string().default('normal'),
      })
      .default({}),
    layout: z.enum(['horizontal', 'vertical']).default('horizontal'),
    slides: z.array(z.string()).default([]),
    styles: z.array(z.string()).default([]),
  });
}

```

HTML template

{% and %} are interpolation characters

```

<!DOCTYPE html>
<html class="hljs">
<head>
<meta charset="utf-8">

```

```

<meta
  name="description"
  content="CodeSlide makes a slideshow for code snippets">
<meta
  name="viewport"
  content="width=device-width, initial-scale=1">
<link
  href="https://fonts.gstatic.com"
  rel="preconnect" crossorigin>
{%~ it.style %}
</head>
<body class="hljs">
<div id="slides">
{%_ for (const [index, slide] of it.slides.entries()) { %}
<div class="slide" id="slide_{%~ index %}">
  {%_ if (index !== 0 && it.layout === 'vertical') { %}
<hr>
  {%_ } %}
{%~ slide %}
</div>
{%_ } %}
</div>
</body>
</html>

```

CSS (Horizontal layout)

```

/*! CodeSlide slides.horizontal.css */
a {
  color: dodgerblue;
}
body {
  margin: 0;
  overflow: hidden;
}
code {
  font-size: 85%;
}
html {
  -webkit-print-color-adjust: exact;
  print-color-adjust: exact;
}

```

```
}
img {
  max-width: 100%;
}
li {
  margin-top: 0.25em;
}
p:empty {
  display: none;
}
pre {
  white-space: pre-wrap;
  overflow-wrap: break-word;
}
pre > code {
  display: block;
  padding: 1em;
}
video {
  max-width: 100%;
}
.slide {
  min-width: calc(100vw - 4em);
  height: calc(100vh - 2em);
  padding: 1em 2em;
  overflow-y: scroll;
  scroll-snap-align: start;
  scroll-snap-stop: always;
  scrollbar-width: none;
}
.slide::-webkit-scrollbar {
  display: none;
}
@media only screen and (max-width: 768px) {
  .slide {
    height: calc(90% - 2em);
    height: calc(100svh - 2em);
  }
}
#slides {
  display: flex;
  flex-direction: row;
  width: 100vw;
  height: 100vh;
  overflow-x: scroll;
  overscroll-behavior: none;
```

```
scroll-behavior: smooth;
scroll-snap-type: x mandatory;
line-height: 1.5;
}
@media print {
  @page {
    margin: 0;
    size: auto;
  }
  #slides {
    width: auto;
    height: auto;
  }
}
```

CSS (Vertical layout)

```
/*! CodeSlide slides.vertical.css */
a {
  color: dodgerblue;
}
body {
  margin: 0;
  overflow: hidden;
}
code {
  font-size: 85%;
}
html {
  -webkit-print-color-adjust: exact;
  print-color-adjust: exact;
}
img {
  max-width: 100%;
}
li {
  margin-top: 0.25em;
}
p:empty {
  display: none;
}
```

```
pre {
  white-space: pre-wrap;
  overflow-wrap: break-word;
}
pre > code {
  display: block;
  padding: 1em;
}
video {
  max-width: 100%;
}
.slide {
  padding: 1em 2em;
}
#slides {
  display: flex;
  flex-direction: column;
  position: absolute; /* fix height on mobile */
  width: 100vw;
  height: 100vh;
  overflow-y: scroll;
  overscroll-behavior: none;
  scroll-behavior: smooth;
  line-height: 1.5;
}
@media print {
  @page {
    margin: 0;
    size: auto;
  }
  #slides {
    width: auto;
    height: auto;
  }
}
```

The entry point

```
import { program } from 'commander';
import { readFileSync, writeFileSync } from 'fs';
import { stdin, stdout } from 'process';
```

```
import { version, homepage, name } from '../package.json';
import { CLIOptions, Manifest } from './schemas';
```

program

```
.name(name)
```

```
.description(`\`
```

```
Example: ${name} -m ./manifest.md -o ./output.html
```

Make a slideshow (HTML/PDF file) for code snippets
with a manifest (Markdown file).

Go to home page for more information: \${homepage}`

```
)
```

```
.version(version, '-v, --version', `
```

```
Check the version number.`
```

```
)
```

```
.helpOption('-h, --help', `
```

```
Check all options and their description.`
```

```
)
```

```
.option('-o, --output [local_path]', `
```

```
The "output file path" of slideshow.
```

```
By default it writes the output to stdout.`
```

```
)
```

```
.option('-m, --manifest [local_path]', `
```

```
The "manifest file path" of slideshow.
```

```
By default it reads manifest from stdin.`
```

```
)
```

```
.action(async (options: CLIOptions) => {
```

```
  let { output, manifest } = CLIOptions.parse(options);
```

```
  const file = output ?? stdout.fd;
```

```
  if (manifest) {
```

```
    const _manifest = await Manifest.parse(
```

```
      readFileSync(manifest, 'utf8')
```

```
    );
```

```
    const { data, encoding } = await Manifest.render(
```

```
      _manifest
```

```
    );
```

```
    writeFileSync(file, data, encoding);
```

```
  } else {
```

```
    let buffer = Buffer.alloc(0);
```

```
    stdin
```

```
      .on('data', (d) => {
```

```
        buffer = Buffer.concat([buffer, d]);
```

```
      })
```

```
      .once('end', async () => {
```

```
        const _manifest = await Manifest.parse(
```



```

        buffer.toString('utf8')
    );
    const { data, encoding } = await Manifest.render(
        _manifest
    );
    writeFileSync(file, data, encoding);
});
}
})
.parseAsync()
.catch((err) => { throw err; }));

```

CLIOptions

- The CLI options schema
- -m, --manifest: Manifest file path
- -o, --output: Output file path

```

import { z } from 'zod';
import { formatZodError } from '../utils';

export type CLIOptions = z.infer<typeof CLIOptions.schema>;

export namespace CLIOptions {
    export const parse = (
        options?: Partial<CLIOptions>
    ): CLIOptions => (
        schema.default({}).parse(options)
    );

    export const schema = z
        .object({
            manifest: z.string().optional(),
            output: z.string().optional(),
        })
        .strict()
        .catch((e) => {
            throw new Error(`\
Cannot parse the CLI options:
\t${formatZodError(e.error.errors[0])}`
        );
    );

```

```
});  
}
```

Miscellaneous

- Because [Node Fetch](#) does not handle file: URI scheme, CodeSlide CLI implements it with `fs.readFileSync`:

```
import fetch from 'node-fetch';  
import statuses from 'statuses';  
import { readFileSync } from 'fs';  
import { pathToFileURL } from 'url';  
  
export const getContent = async (  
  path: string | URL,  
) : Promise<string> => {  
  if (typeof path === 'string') {  
    try {  
      path = new URL(path);  
    } catch (err) {  
      path = pathToFileURL(path.toString());  
    }  
  }  
  if (path.protocol === 'file:') {  
    return readFileSync(path, 'utf8');  
  } else {  
    const res = await fetch(path);  
    if (res.ok) { return res.text(); }  
    const { status, url } = res;  
    throw new Error(  
      `Cannot GET ${url} due to the error ${status}  
      (${statuses(status)})`  
    );  
  }  
};
```

Thanks for your watching!

- The repository of this example is [here](#)
- See other CodeSlide CLI examples [here](#)
- See the installation guide of CodeSlide CLI [here](#)