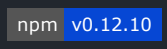
Behind the Scenes of CodeSlide CLI

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

CodeSlide CLI



Usage demo

```
<yilan 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 `Renderer` schema
- `Manifest.parse`: Parse a manifest file into a `Manifest` schema
- `Manifest.print`: Render the slideshow and print it to the output

```
import { PathOrFileDescriptor, writeFileSync } from 'fs';
import matter from 'gray-matter';
import { launch } from 'puppeteer';
import { FrontMatter } from './FrontMatter';
import { SlideShow } from './SlideShow';
import { Renderer } from '../../../../src';
import { getContent } from '../utils';

export type Manifest = FrontMatter & SlideShow;

export namespace Manifest {
  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 = FrontMatter.parse(codeslide);
```

```
    fm.styles = await Promise.all(
      fm.styles.map((path) => getContent(path))
    );
    const slides = await SlideShow.parse(content);
    return { ...fm, ...slides };
  };

  export const print = async (
    output: PathOrFileDescriptor,
    manifest: Manifest,
  ): Promise<void> => {
    if (manifest.format === 'html') {
      writeFileSync(output, Renderer.render(manifest), 'utf8');
    } else if (manifest.format === 'pdf') {
      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,
      });
      const closeBrowser = browser.close();
      writeFileSync(output, result, 'base64');
      await closeBrowser;
    }
  };
}
```

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 { z } from 'zod';
import { Renderer } from '../../../../src';
import { version } from '../../../../package.json';
import { formatZodErrors } from '../utils';

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

```
export namespace FrontMatter {
  export const parse = (
    fm?: Partial<FrontMatter>
  ): FrontMatter => {
    const result = schema.default({}).safeParse(fm);
    if (! result.success) {
      throw new Error(
        `Cannot parse the Front Matter section: ${
          formatZodErrors(result.error.errors)
        }`
      );
    }
    return result.data;
  };

  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().default(version),
    })
    .and(Renderer.schema.omit({ slides: true }))
    .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

```ts
import hljs 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: (code, language) => {
    return hljs.highlight(code, { language }).value;
  },
  walkTokens: async (token: marked.Token) => {
    if (token.type === 'link') {
      const { href, text, raw } = token;
      if (! text.startsWith(':')) {
        return;
      }
      const [prefix, suffix] = <[string, string | undefined]>
        text.split('.');
      if (prefix === ':slide') {
        token = _toHTMLToken(token);
        token.raw = raw;
        token.text = await getContent(href)
          .then((content) => _parseMarkdown(content));
      } else if (prefix === ':code') {
        token = _toHTMLToken(token);
        token.raw = raw;
        const code = await getContent(href).then((content) => (
          hljs.highlight(content, {
            language: suffix ?? 'plaintext'
          })
        ));
        token.text = `\
<pre><code${
  code.language ? ` class="language-${code.language}"` : ''
}>${
  code.value
}</code></pre>`;
      }
    }
```

```
  },
});

const _toHTMLToken = (
  token: marked.Token
): marked.Tokens.HTML => {
  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 = 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 GithubDarkDimmed from './github-dark-dimmed.css';
import HorizontalStylesheet from './slides.horizontal.css';
import VerticalStylesheet from './slides.vertical.css';
import Template from './slides.html';

const Stylesheets = {
  horizontal: HorizontalStylesheet,
  vertical: VerticalStylesheet,
  highlight: GithubDarkDimmed,
```

```
};

export { Stylesheets, Template };


import { render as renderEta } from 'eta';
import { z } from 'zod';
import { Stylesheets, Template } 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 => {
    return renderEta(Template, {
      layout: renderer.layout,
      slides: renderer.slides,
      style: `\
<style>
${[
  Stylesheets['highlight'],
  Stylesheets[renderer.layout],
  ...renderer.styles,
`code {
  font-family: ${renderer.fontFamily};
  font-size: 85%;
}`,
`#slides {
  font-family: system-ui;
  font-size: ${renderer.fontSize};
  font-weight: ${renderer.fontWeight};
  line-height: 1.5;
}`,
].join('\n')}
</style>`,
    },
    {
      autoTrim: false,
      tags: ['{%', '%}']
    });
  };

  export const schema = z
    .object({
      fontFamily: z.string().optional().transform((arg) => `\
${arg ? `${arg}, ` : ''}ui-monospace, SFMono-Regular, \
SF Mono, Menlo, Consolas, Liberation Mono, monospace`
```

```
      ),
      fontSize: z.string().default('large'),
      fontWeight: z.string().default('normal'),
      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">
{%~ 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 */
html, body {
  margin: 0;
```

```
  -webkit-print-color-adjust: exact;
  print-color-adjust: exact;
  overflow: hidden;
}
a {
  color: dodgerblue;
}
li {
  margin-top: 0.25em;
}
p:empty {
  display: none;
}
pre {
  white-space: pre-wrap;
  overflow-wrap: break-word;
}
pre > code {
  display: block;
  padding: 1em;
}
.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;
}
@media print {
  @page {
    margin: 0;
    size: auto;
  }
```

```
  #slides {
    width: auto;
    height: auto;
  }
}
```

CSS (Vertical layout)

```
/*! CodeSlide slides.vertical.css */
html, body {
  margin: 0;
  -webkit-print-color-adjust: exact;
  print-color-adjust: exact;
  overflow: hidden;
}
a {
  color: dodgerblue;
}
li {
  margin-top: 0.25em;
}
p:empty {
  display: none;
}
pre {
  white-space: pre-wrap;
  overflow-wrap: break-word;
}
pre > code {
  display: block;
  padding: 1em;
}
.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;
}
@media print {
  @page {
    margin: 0;
    size: auto;
```

```
  }
  #slides {
    width: auto;
    height: auto;
  }
}
```

The entry point

```
import { program } from 'commander';
import { readFileSync } 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);
    if (manifest) {
      const _manifest = await Manifest.parse(
        readFileSync(manifest, 'utf8')
      );
      await Manifest.print(output ?? stdout.fd, _manifest);
    } else {
      let data = Buffer.alloc(0);
      stdin
```

```
        .on('data', (d) => {
          data = Buffer.concat([data, d]);
        })
        .once('end', async () => {
          const _manifest = await Manifest.parse(
            data.toString('utf8')
          );
          await Manifest.print(output ?? stdout.fd, _manifest);
        });
    }
  })
  .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 { formatZodErrors } from '../utils';

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

export namespace CLIOptions {
  export const parse = (
    options?: Partial<CLIOptions>
  ): CLIOptions => {
    const result = schema.default({}).safeParse(options);
    if (! result.success) {
      throw new Error(
        `Cannot parse the CLI options: ${
          formatZodErrors(result.error.errors)
        }`
      );
    }
    return result.data;
  };

  export const schema = z
    .strictObject({
      manifest: z.string().optional(),
      output: z.string().optional(),
    });
}
```

Miscellaneous

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

```ts
import fetch from 'node-fetch';
import status 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 {
    return fetch(path).then(async (r) => {
      if (r.ok) { return r.text(); }
      throw new Error(
        `Cannot GET ${r.url}: "${r.status} ${status(r.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