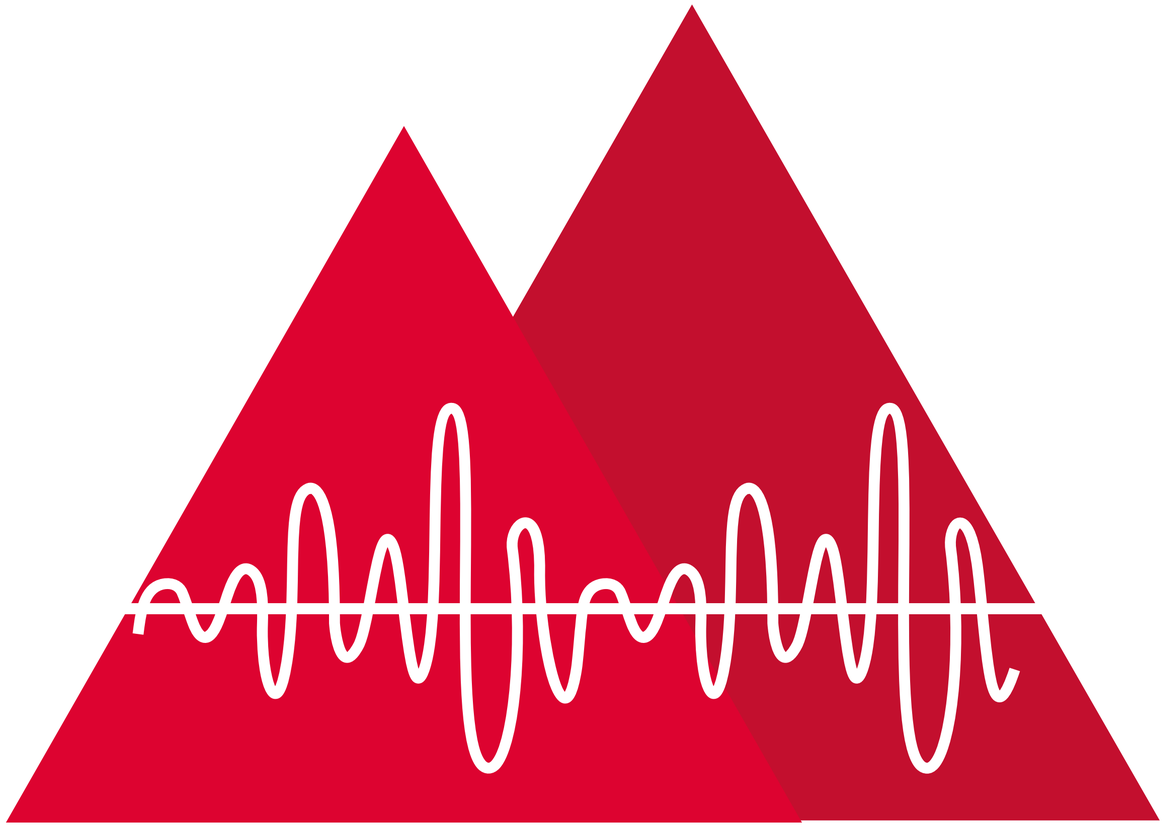
Analog

A meta framework for Angular



What is Analog?

What is a meta framework?

- A framework built on top of another framework? Okay???
- Why?
  - Cool ~~shit~~ like file based routing and analog single file components (sfc)
  - You may have heard of Next.js, SvelteKit, and Nuxt
- Nice

What is Analog?

So why Analog?

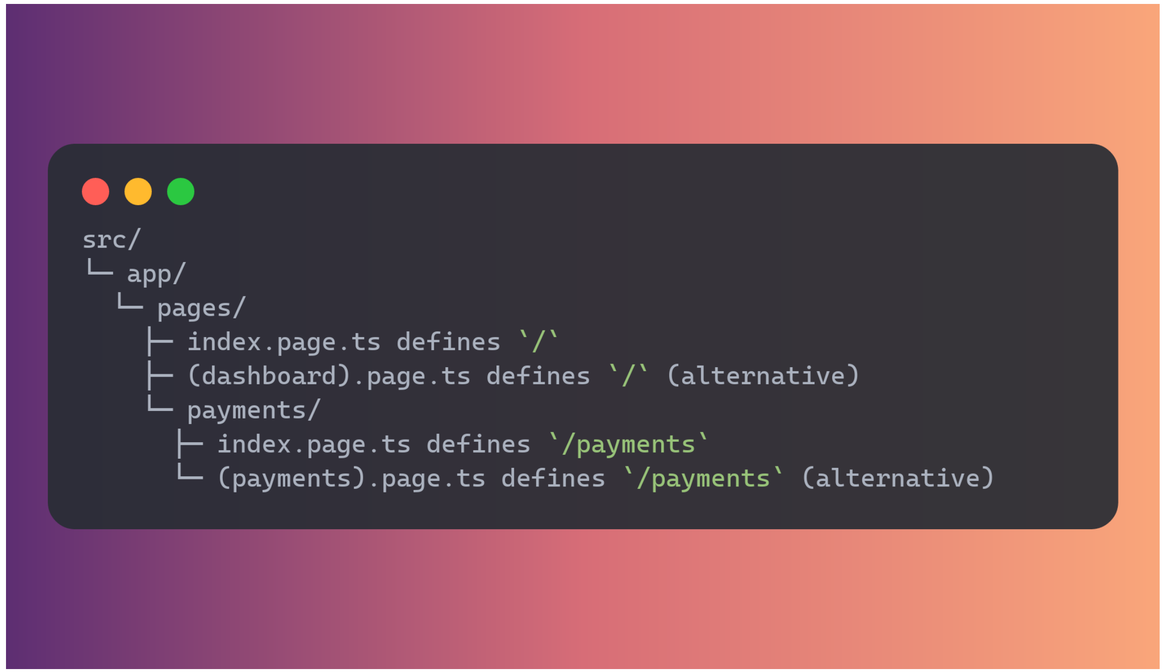
- Angular is dope
- Why not make it better?
- Honestly tho
  - File based routing
  - Content routes (markdown routes)
  - Server side rendering / static site generation
  - .analog component authoring

File Based Routing

File Based Routing

Index Routes

- Index routes use the filename to define the route
- Either with `index.page.ts` or filename as the route path enclosed in parenthesis `(dashboard).page.ts`
- Can be nested as well



File based routing

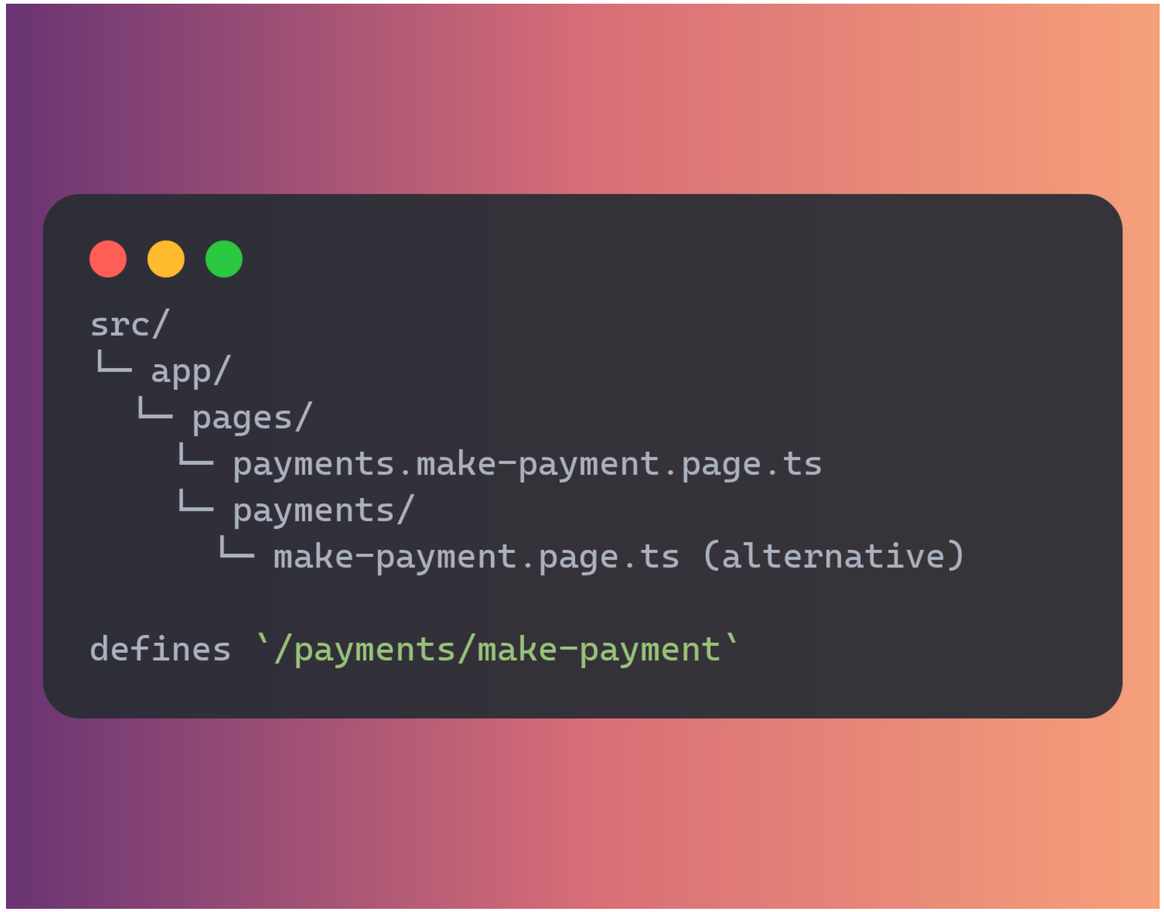
Static Routes

- Static routes are defined by using the filename as the route path
- Think `/pages/payments.page.ts` defines `/payments`
- We also have `/pages/payments/index.page.ts`
- Or `/pages/payments/(payments).page.ts`

File based routing

Nesting Static Routes

- Nesting route files in folders
  - `/pages/payments/make-payment.page.ts`
  - Or dot notation `/pages/payments.make-payment.page.ts`



File based routing

Route Groups

- Routes can be grouped together in the same folder without adding a route path segment by wrapping a folder name in parenthesis
- `/pages/(auth)/login.page.ts` defines `/login`

File based routing

Dynamic Routes

- Dynamic routes are defined by using the filename as the route path enclosed in square brackets
- The parameter for the route is extracted from the route path

File based routing

Dynamic Routes

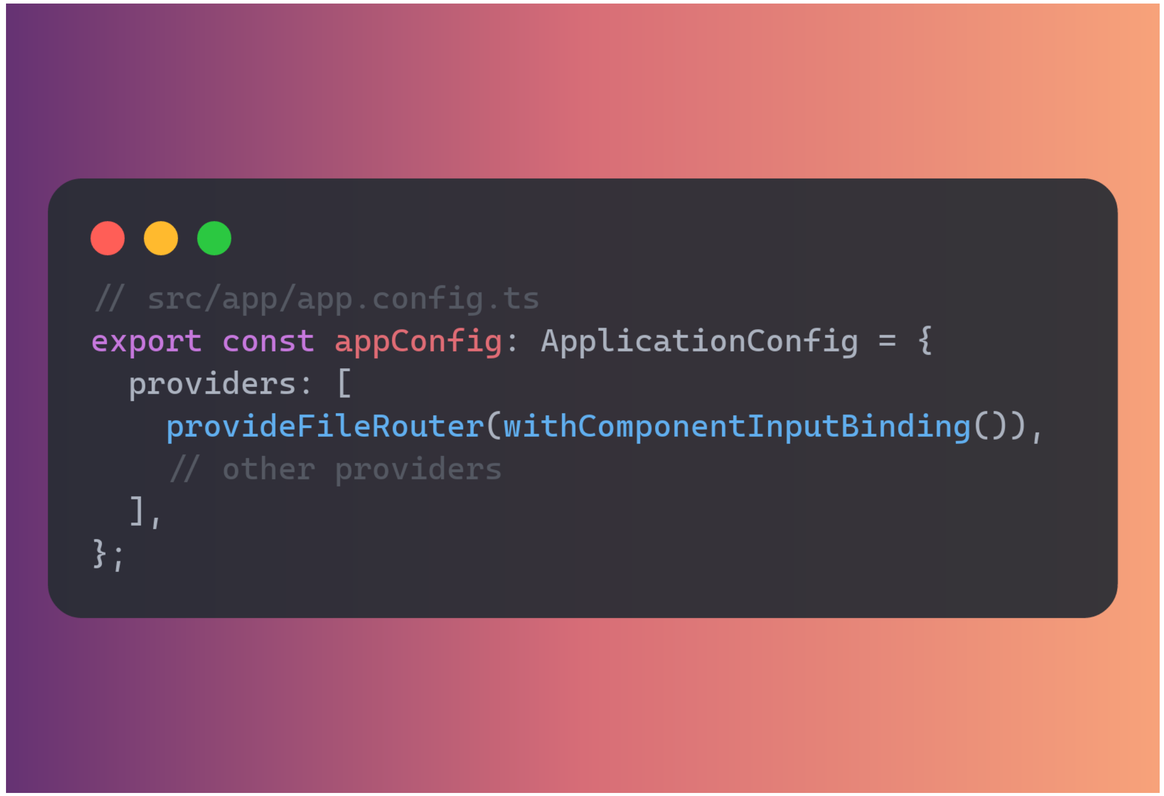
- The example route in `src/app/pages/payments/[paymentId].page.ts` defines a `/payments/:paymentId` route
- Can also be accessed as an input

```ts
import { Component, inject } from '@angular/core';
import { AsyncPipe } from '@angular/common';
import { ActivatedRoute } from '@angular/router';
import { map } from 'rxjs';

@Component({
  standalone: true,
  imports: [AsyncPipe],
  template: `
    <h2>Payment Details</h2>

    ID: {{ paymentId$ | async }}
  `,
})
export default class PaymentDetailsPage {
  private readonly route = inject(ActivatedRoute);

  readonly paymentId$ = this.route.paramMap.pipe(
    map((params) => params.get('paymentId')),
  );
}
```

File based routing

Dynamic Routes with Component Input

- First, add the `withComponentInputBinding()` to the arguments for the `provideFileRouter()` function in `app.config.ts`



```ts
// src/app/app.config.ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideFileRouter(withComponentInputBinding()),
    // other providers
  ],
};
```

File based routing

Dynamic Routes with Component Input

```ts
import { Component, Input } from '@angular/core';

@Component({
  standalone: true,
  template: `
    <h2>Payment Details</h2>

    ID: {{ paymentId }}
  `,
})
export default class PaymentDetailsPage {
  @Input() paymentId: string | null = null;
}
```

File based routing

Layout Routes

- Layout routes are defined by using a parent file and child folder with the same name
- The following structure below represents a layout route

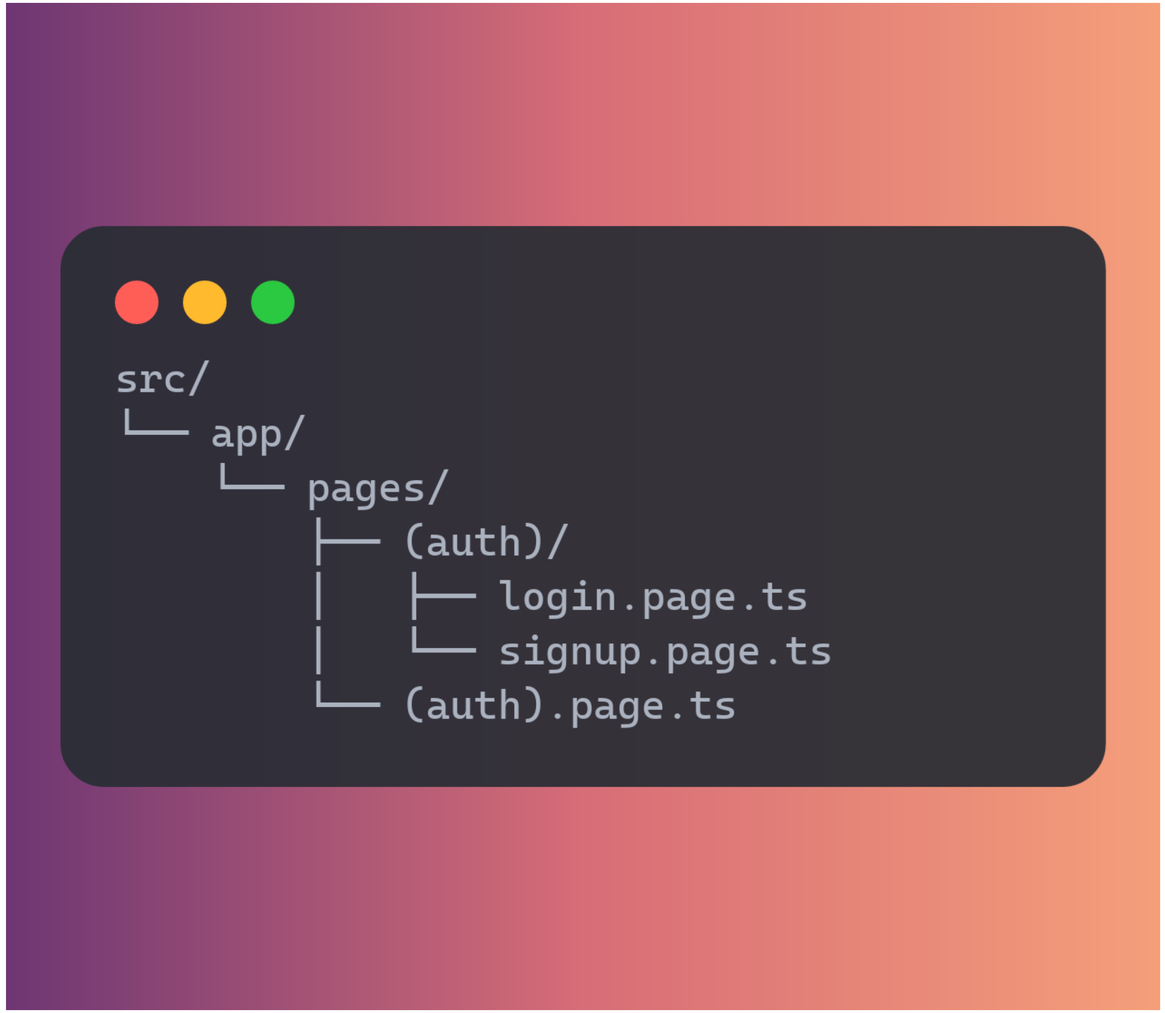
File based routing

Layout Routes

```ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  standalone: true,
  imports: [RouterOutlet],
  template: `
    <h2>Payments</h2>

    <router-outlet></router-outlet>
  `,
})
export default class PaymentsPage {}
```

File based routing

Pathless Layout Routes



File based routing

Catch All Route

- Catch-all routes are defined by using the filename as the route path prefixed with 3 periods enclosed in square brackets
- `src/app/pages/[…not-found].page.ts` defines wildcard `**`
- This route is usually for 404 pages.

Filie based routing

Catch All Route

- Defined using the filename as the route path prefixed with 3 periods enclosed in square brackets
- `[…not-found].page.ts` defines wildcard `**`

File based routing

Altogether

Content Routes

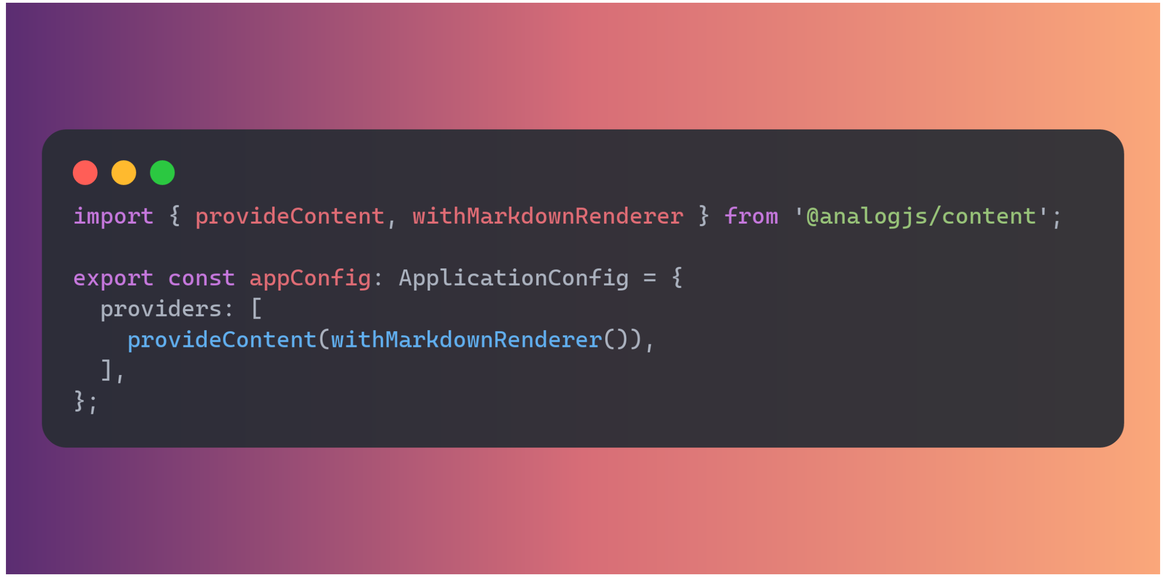
Content Routes

Markdown Content Routes

- Analog also supports using markdown content as routes, and rendering markdown content in components
- This can be useful for text have routes that don’t rely on data processing

Content Routes

app.config.ts

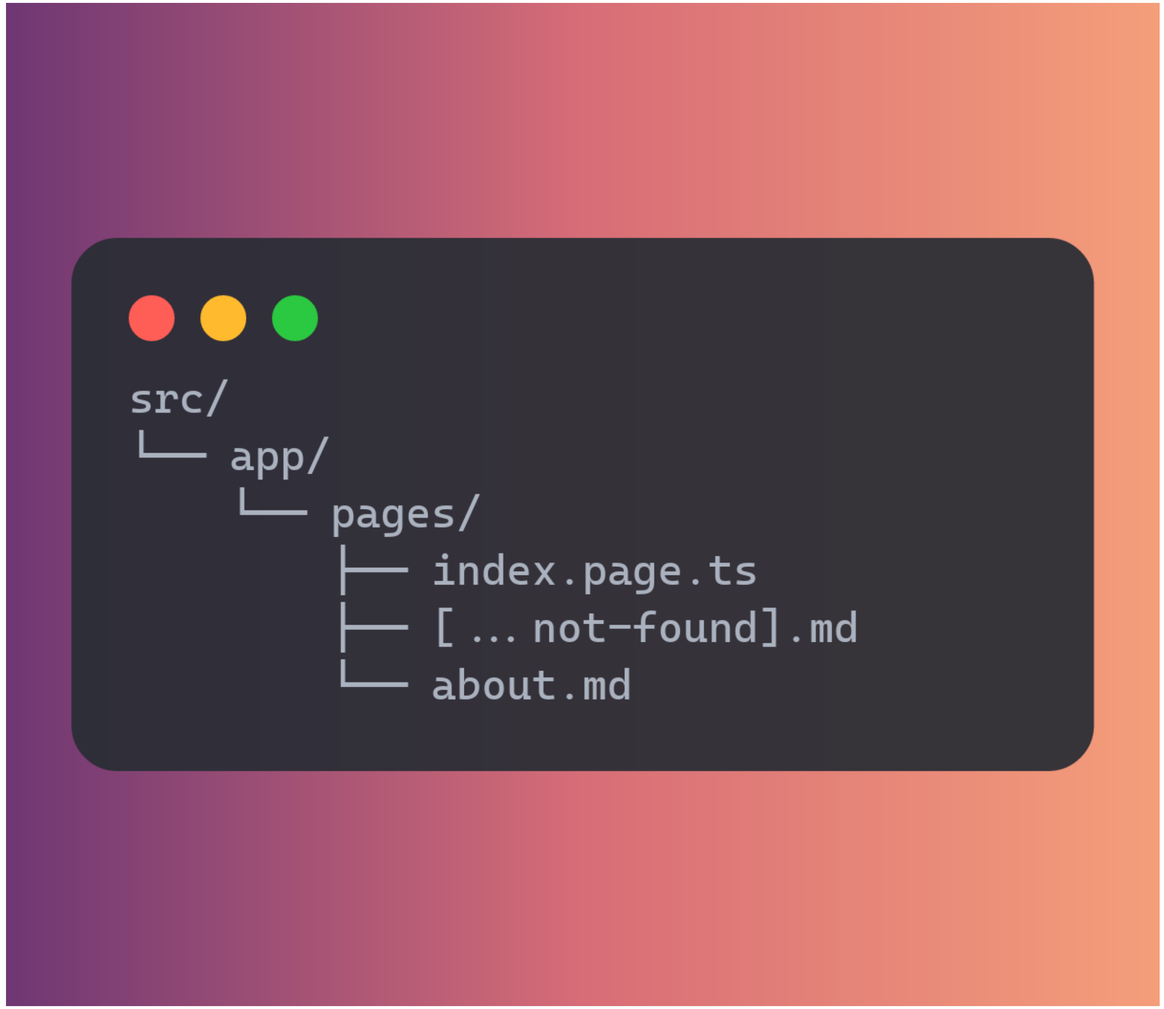


```ts
import { provideContent, withMarkdownRenderer } from '@analogjs/content';

export const appConfig: ApplicationConfig = {
  providers: [
    provideContent(withMarkdownRenderer()),
  ],
};
```

Content Routes

Defining the content route

- Content routes include support for frontmatter, metatags, and syntax highlighting with PrismJS
- Creating `src/app/pages/about.md` defines an `/about` route



Content Routes

Defining Content Files

- Markdown content files can be provided in the `src/content/` folder
- Here you can list markdown files such as help documents or blog posts

Content Routes

Using the Content Files List

- To get a list using the list of content files in the `src/content/` folder, use the

```
injectContentFiles<Attributes>(
)
```

Content Routes

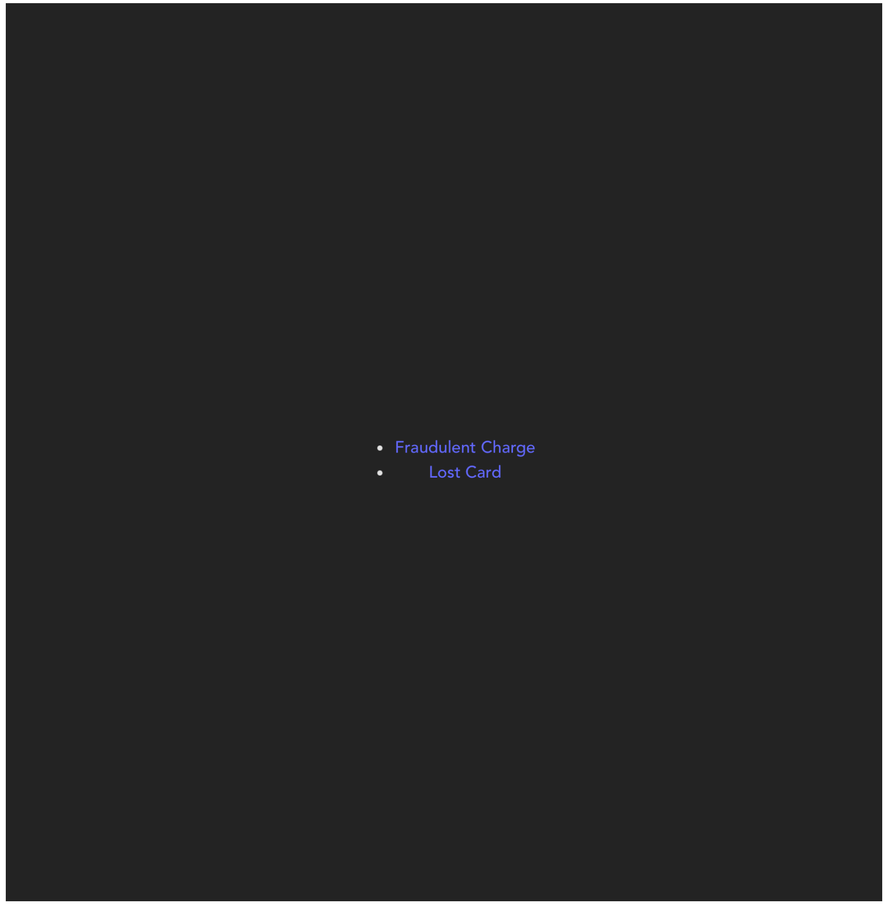
`fraudulent-charge.md`

Content Routes

- You can filter the files using a `filterFn` constructed from `InjectContentFilesFilterFunction<T>`
- What?

Content Routes

Using the Content Files List




```ts
// src/app/pages/help/(message-list).page.ts

import { Component } from '@angular/core';
import { RouterLink } from '@angular/router';
import { injectContentFiles } from '@analogjs/content';

export interface MessageAttributes {
  title: string;
  slug: string;
  description: string;
  coverImage: string;
}

@Component({
  standalone: true,
  imports: [RouterLink],
  template: `
    <ul>
      @for (message of messages; track message.slug) {
        <li>
          <a [routerLink]="['/help', message.slug]">{{
            message.attributes.title
          }}</a>
        </li>
      }
    </ul>
  `,
})
export default class MessagesListPage {
  readonly messages = injectContentFiles<MessageAttributes>((contentFile) =>
    contentFile.filename.includes('/src/content/help/'),
  );
}
```

Content Routes

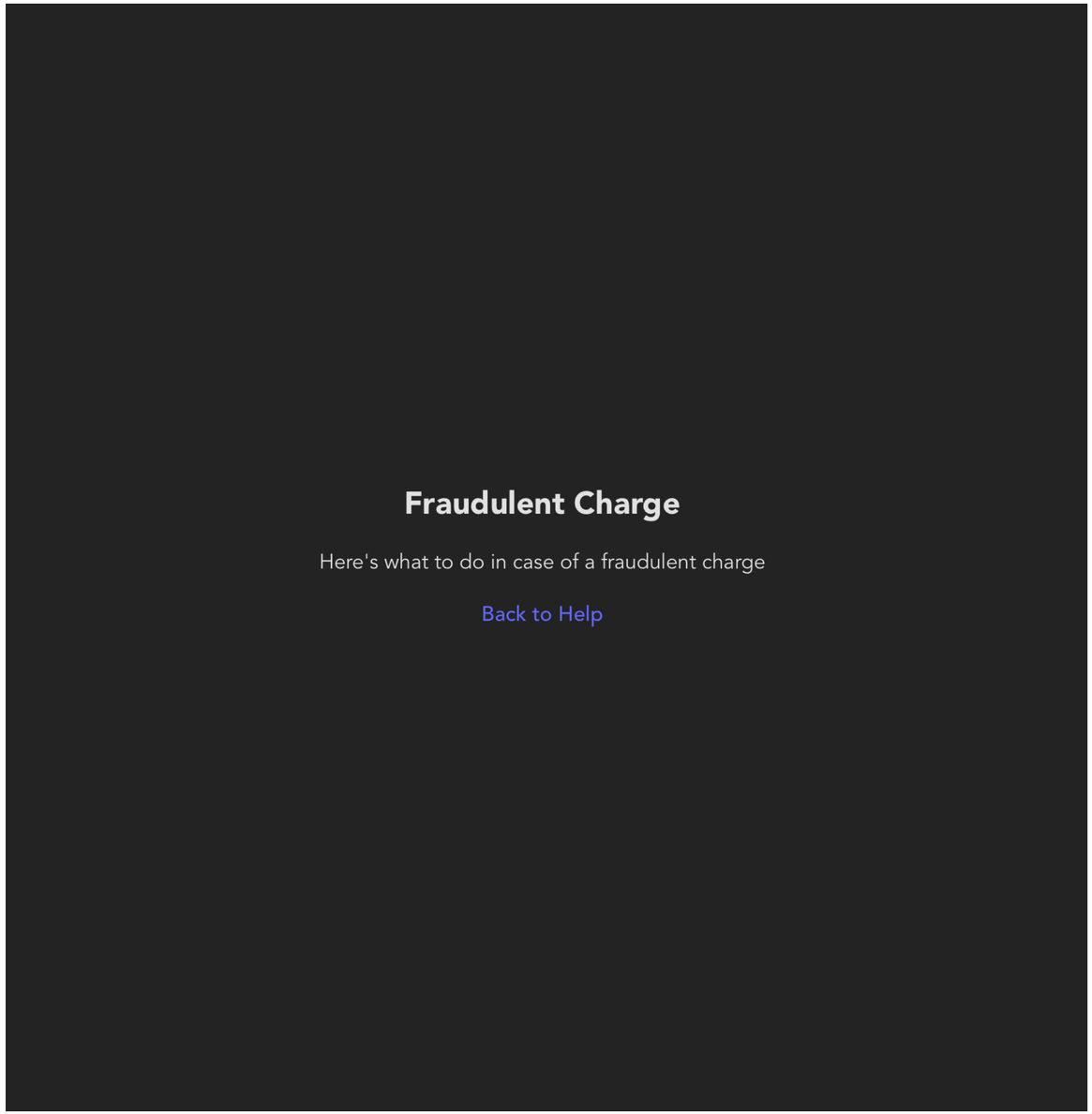
Using the Analog Markdown Component

- Analog provides a `MarkdownComponent` and `injectContent()` function for rendering markdown content with frontmatter
- The `injectContent()` function uses the `slug` route parameter by default to get the content file from the `src/content/` folder

```ts
// src/app/pages/help/[slug].page.ts
import { injectContent, MarkdownComponent } from '@analogjs/content';
import { AsyncPipe } from '@angular/common';
import { Component } from '@angular/core';
import { MessageAttributes } from './(messages-list).page';

@Component({
  standalone: true,
  imports: [MarkdownComponent, AsyncPipe],
  template: `
    @if (message$ | async; as message) {
      <ng-container>
        <analog-markdown [content]="message.content" />
      </ng-container>
    }
  `,
})
export default class MessagePage {
  readonly message$ = injectContent<MessageAttributes>({
    param: 'slug',
    subdirectory: 'help',
  });
}
```

Content Routes

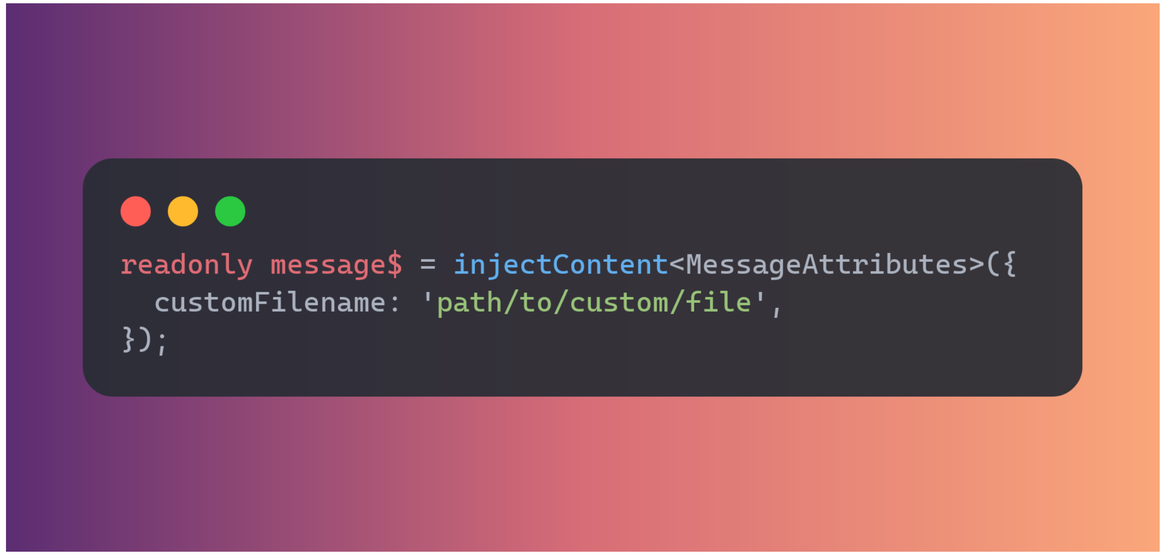
`/help/fraudulent-charge` **route**



Content Routes

Loading Custom Content

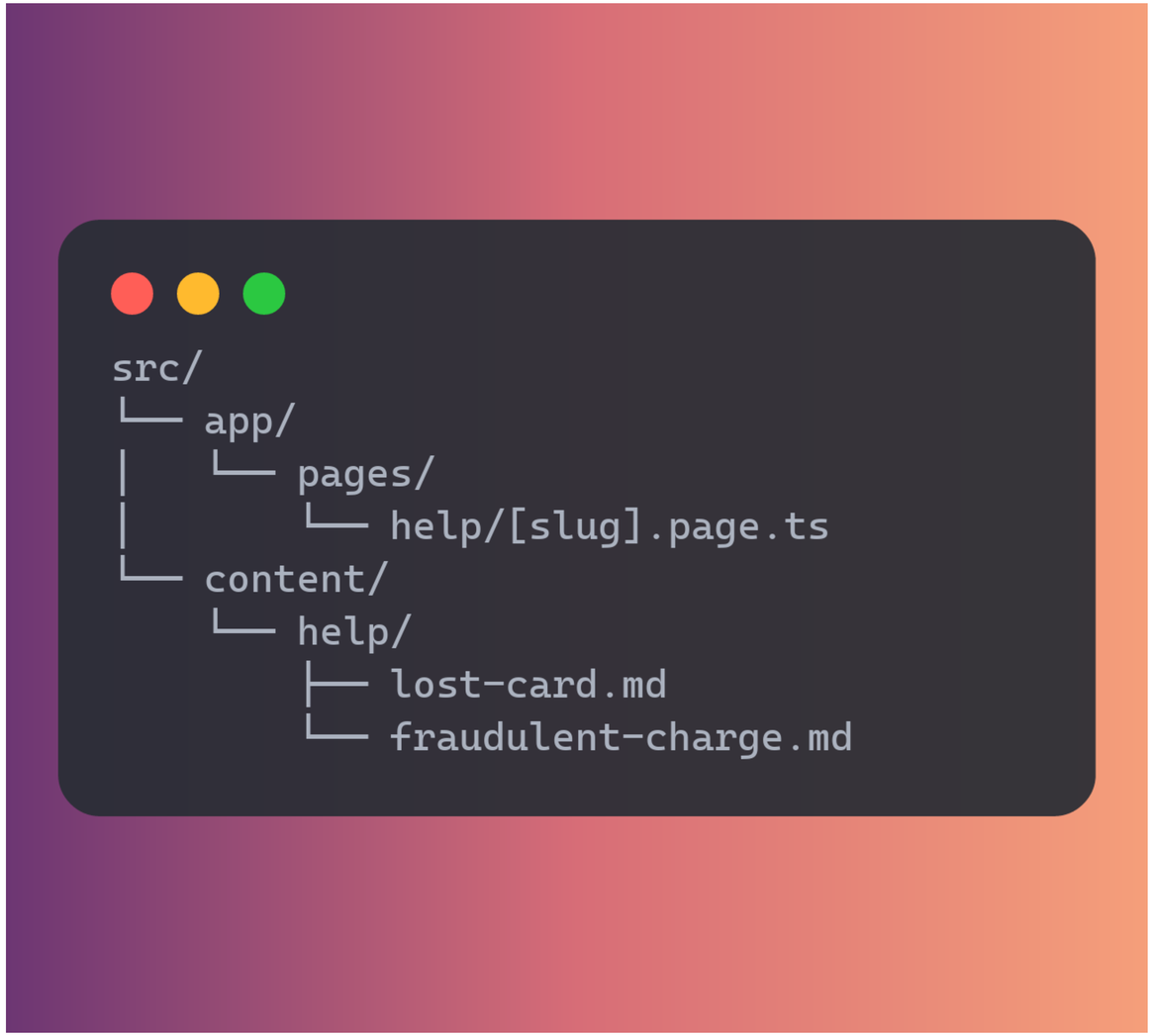
- By default, Analog uses the route params to build the filename for retrieving a content file from the `src/content/ folder`
- The `injectContent()` function can be used by passing an object that contains the `customFilename` property or object with `param` and `subdirectory` properties



Content Routes

Support for content subdirectories

- Analog also supports subdirectories within your content folder.
- The `injectContent` function can also be used with an object that contains the route parameter and subdirectory name.
- This can be useful if, for instance, you have blog posts, as well as a portfolio of project markdown files to be used on the site.



API ROUTES

API ROUTES

Defining an API Routes

- Analog supports defining API routes that can be used to serve data to the application
- API routes are defined in the `src/server/routes` folder
- API routes are also filesystem based, and are exposed under the default `/api` prefix in development
- `src/server/routes/v1/hello.ts` creates api endpoint `/api/v1/hello`

API ROUTES

DYNAMIC API ROUTES

- Dynamic API routes are defined by using the filename as the route path enclosed in square brackets
- Parameters can be accessed via `event.context.params`

API ROUTES

DYNAMIC API ROUTES

- Can also use `getRouterParam` instead of `event.context.params`

```ts
// /server/routes/v1/hello/[name].ts
import { defineEventHandler, getRouterParam } from "h3";

export default defineEventHandler((event) => {
  const name = getRouterParam(event, "name");
  return `Hello, ${name}!`;
});
```

API ROUTES

HTTP Requests & Error Handling

- File names can be suffixed with `.get`, `.post`, `.put`, `.delete`, etc. to match the specific HTTP request method
- If no errors are thrown, a status code of `200 OK` will be returned
- Any uncaught errors will return a `500 Internal Server Error`
- To return other error codes, throw an exception with `createError`
- Use `readBody` for POST body and `getQuery` for query params

```ts
// /server/routes/v1/transactions/[id].get.ts
import { createError, defineEventHandler, getRouterParam } from 'h3';

export default defineEventHandler(async (event) => {
  const id = getRouterParam(event, 'id');
  if (!idIsValid(id)) {
    throw createError({
      statusCode: 400,
      statusMessage: 'ID invalid',
    });
  }

  // NOTE: here we would fetch transaction from db

  const transaction = await fetchTransactionWithId(id);
  return transaction;
});
```

SERVER SIDE DATA FETCHING

SERVER SIDE DATA FETCHING

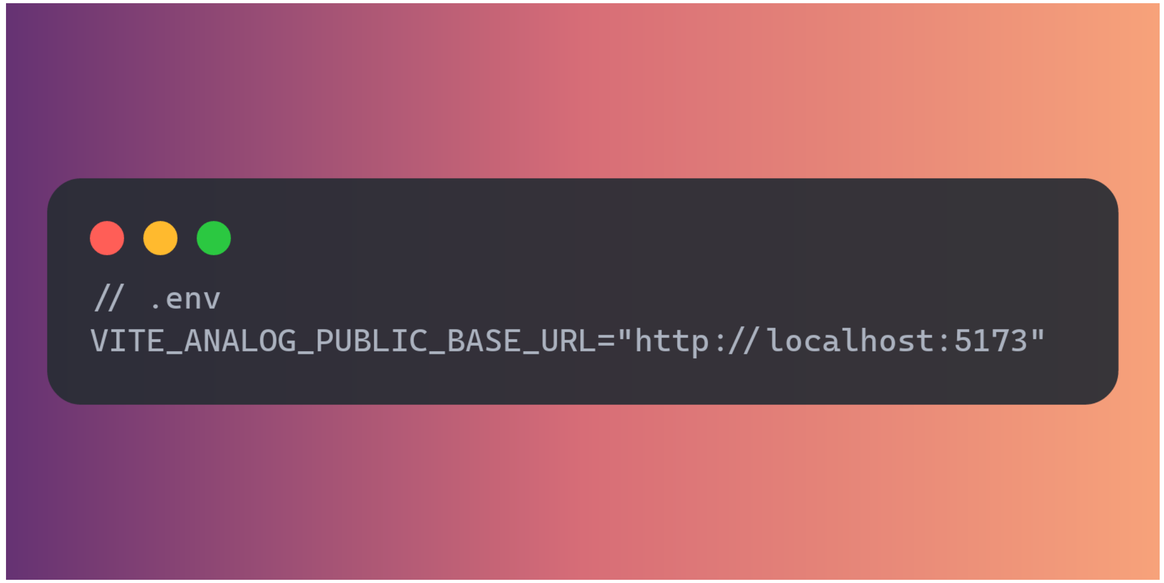
File Structure

- Analog supports fetching data from the server before loading a page
- To fetch the data from the server, create a `.server.ts` file that contains the async load function alongside the `.page.ts` file

SERVER SIDE DATA FETCHING

STEPS TO ENABLE

- Analog requires the public base URL to be set when using the server-side data fetching
- Set an environment variable, using a `.env` file to define the public base URL
- The environment variable must also be set when building for deployment



SERVER SIDE DATA FETCHING

FETCH THE DATA

```ts
// src/app/pages/index.server.ts
import { PageServerLoad } from '@analogjs/router';

export const load = async ({
  params, // params/queryParams from the request
  req, // H3 Request
  res, // H3 Response handler
  fetch, // internal fetch for direct API calls,
  event, // full request event
}: PageServerLoad) => {
  // we'd generally return data from a DB or API call here
  return {
    loaded: true,
  };
};
```

SERVER SIDE DATA FETCHING

Access the data in component using `injectLoad`

```ts
// src/app/pages/index.page.ts
import { Component } from "@angular/core";
import { toSignal } from "@angular/core/rxjs-interop";
import { injectLoad } from "@analogjs/router";

import { load } from "./index.server"; // not included in client build

@Component({
  standalone: true,
  template: `
    <h2>Dashboard</h2>

    Loaded: {{ data().loaded }}
  `,
})
export default class DashboardComponent {
  data = toSignal(injectLoad<typeof load>(), { requireSync: true });
}
```

SERVER SIDE DATA FETCHING

ACCESSING DATA AS AN INPUT

- Update `app.config.ts` with `provideFileRouter(withComponentInputBinding())`

SERVER SIDE DATA FETCHING

ACCESSING DATA AS AN INPUT

```ts
// src/app/pages/index.page.ts
import { Component, Input } from "@angular/core";
import { LoadResult } from "@analogjs/router";

import { load } from "./index.server"; // not included in client build

@Component({
  standalone: true,
  template: `
    <h2>Dashboard</h2>
    Loaded: {{ data.loaded }}
  `,
})
export default class DashboardComponent {
  @Input() load(data: LoadResult<typeof load>) {
    this.data = data;
  }

  data!: LoadResult<typeof load>;
}
```

Server Side Rendering (SSR)

SERVER SIDE RENDERING

WHAT IS IT???

- When SSR is on (enabled by default) `/` is pre-rendered
- When SSR is enabled the root must be pre-rendered
- When the content is pre-rendered the server takes the majority of processing power instead of the client (web browser) which leads to improved (faster) experience for the user and search engines are able to index it because the client is fed the rendered page

SERVER SIDE RENDERING

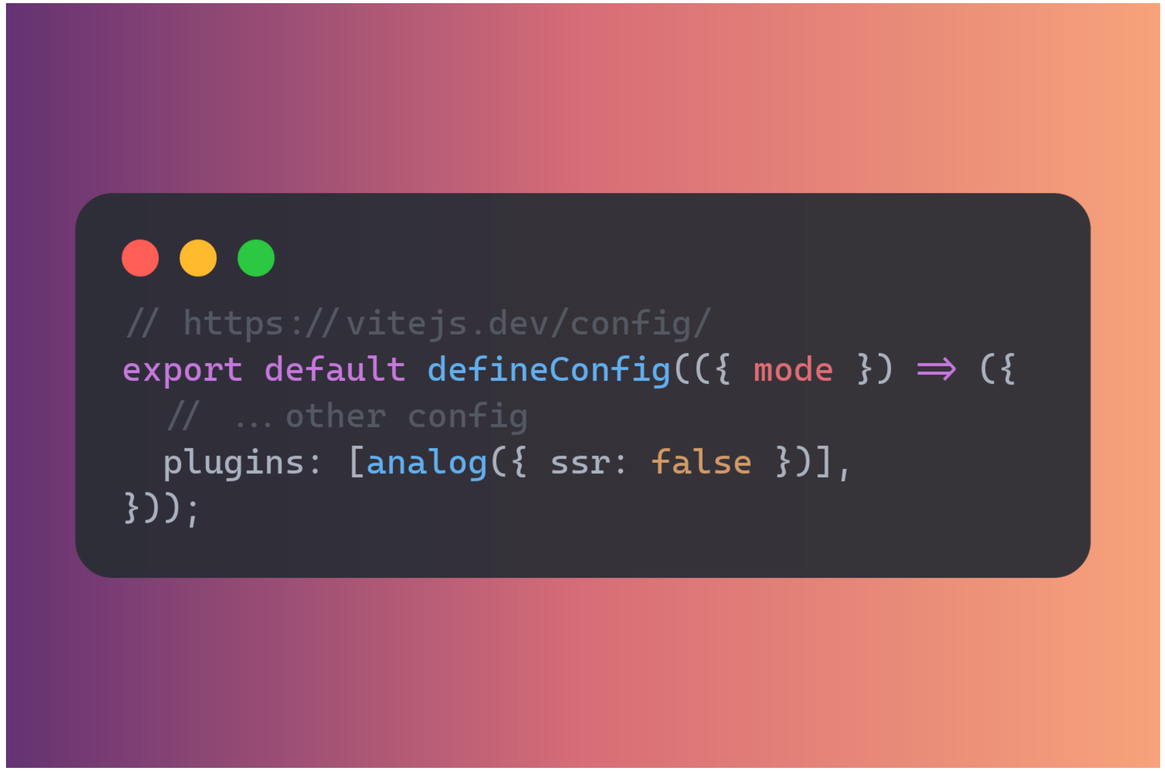
Pre-rendering Routes

- With SSR, the "/" route is pre-rendered by default.
- It is a necessary step to return a rendered HTML when the user visits the root of the application
- The pre-rendered routes can be customized, but keep in mind to include the "/" route also.

```
// vite.config.js
import { defineConfig } from "vite";
import analog from "@analogjs/platform";
// https://vitejs.dev/config/
export default defineConfig(({ mode }) => ({
  // ...other config
  plugins: [
    analog({
      prerender: {
        routes: ["/", "/about"],
      },
    }),
  ],
}));
```

SERVER SIDE RENDERING

Can be disabled



```
// https://vitejs.dev/config/
export default defineConfig(({ mode }) => ({
  // ...other config
  plugins: [analog({ ssr: false })],
}));
```

Static Site Generation (SSG)

STATIC SITE GENERATION

What Is SSG?

- Pre-rendering provided routes to static HTML files along with the client-side application
- Cool because then the browser doesn’t need to load a bunch of JS to render your website = lighter weight, requires less client side processing, and less power
- Webcrawlers can then also access your webpage as well through sitemap generation
- Use the `prerender` property in `vite.config.ts` to configure routes to be rendered at build time
- HTML pre-rendered and then rehydrated with client side interactivity. JS is used for rehydration to attach event listeners, initialize components, and make the content interactive.

STATIC SITE GENERATION

vite.config.ts

```ts
import { defineConfig } from 'vite';
import analog from '@analogjs/platform';

// https://vitejs.dev/config/
export default defineConfig(({ mode }) => ({
  // ...other config
  plugins: [
    analog({
      prerender: {
        routes: async () => ['/', '/about', '/contact'],
      },
    }),
  ],
}));
```

Static Site Generation

From Content Routes

- Directory structure may not be reflected 1:1 in your apps path
- Must pass a `transform` function which maps the file paths to the URLs

```ts
// vite.config.ts
import { defineConfig } from "vite";
import analog, { type PrerenderContentFile } from "@analogjs/platform";

// https://vitejs.dev/config/
export default defineConfig(({ mode }) => ({
  plugins: [
    analog({
      prerender: {
        routes: async () => [
          "/",
          "/about",
          "/contact",
          {
            contentDir: "src/content/help",
            transform: (file: PrerenderContentFile) => {
              // do not include files marked as draft in frontmatter
              if (file.attributes.draft) {
                return false;
              }
              // use the slug from frontmatter if defined, otherwise use the files basename
              const slug = file.attributes.slug || file.name;
              return `/help/${slug}`;
            },
          },
        ],
      },
    }),
  ],
}));
```

Static Site Generation
SITEMAP GENERATION

- Analog also supports automatic sitemap generation
- Analog generates a sitemap in the `dist/analog/public` directory when running a build if a sitemap configuration is provided
- Further improved indexing and SEO

Static Site Generation

Post-Rendering Hooks

- Post-rendering hooks during the pre-rendering process which allows for inlining critical css or modifying HTML files

```ts
// vite.config.ts
import analog from '@analogjs/platform';
import { defineConfig } from 'vite';
import { PrerenderRoute } from 'nitropack';

// https://vitejs.dev/config/
export default defineConfig(() => {
  return {
    publicDir: 'src/public',
    build: {
      target: ['es2020'],
    },
    plugins: [
      analog({
        static: true,
        prerender: {
          routes: async () => [],
          postRenderingHooks: [
            async (route: PrerenderRoute) => console.log(route),
          ],
        },
      }),
    ],
  };
});
```

SSG

Post-rendering gTag

```ts
// vite.config.ts
/// <reference types="vitest" />

import analog from '@analogjs/platform';
import { defineConfig } from 'vite';
import { PrerenderRoute } from 'nitropack';

// https://vitejs.dev/config/
export default defineConfig(() => {
  return {
    publicDir: 'src/public',
    build: {
      target: ['es2020'],
    },
    plugins: [
      analog({
        static: true,
        prerender: {
          routes: async () => ['/', '/about'],
          postRenderingHooks: [
            async (route: PrerenderRoute) => {
              const gTag = `<script>
              (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
                (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
                m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
                })(window,document,'script','//www.google-analytics.com/analytics.js','ga');

                ga('create', 'UA-xxxxxx-1', 'auto');
                ga('send', 'pageview');
              </script>`;
              if (route.route === '/about') {
                route.contents = route.contents?.concat(gTag);
              }
            },
          ],
        },
      }),
    ],
  };
});
```

.analog SFC (experimental!!)

.analog SFC

Current SFC

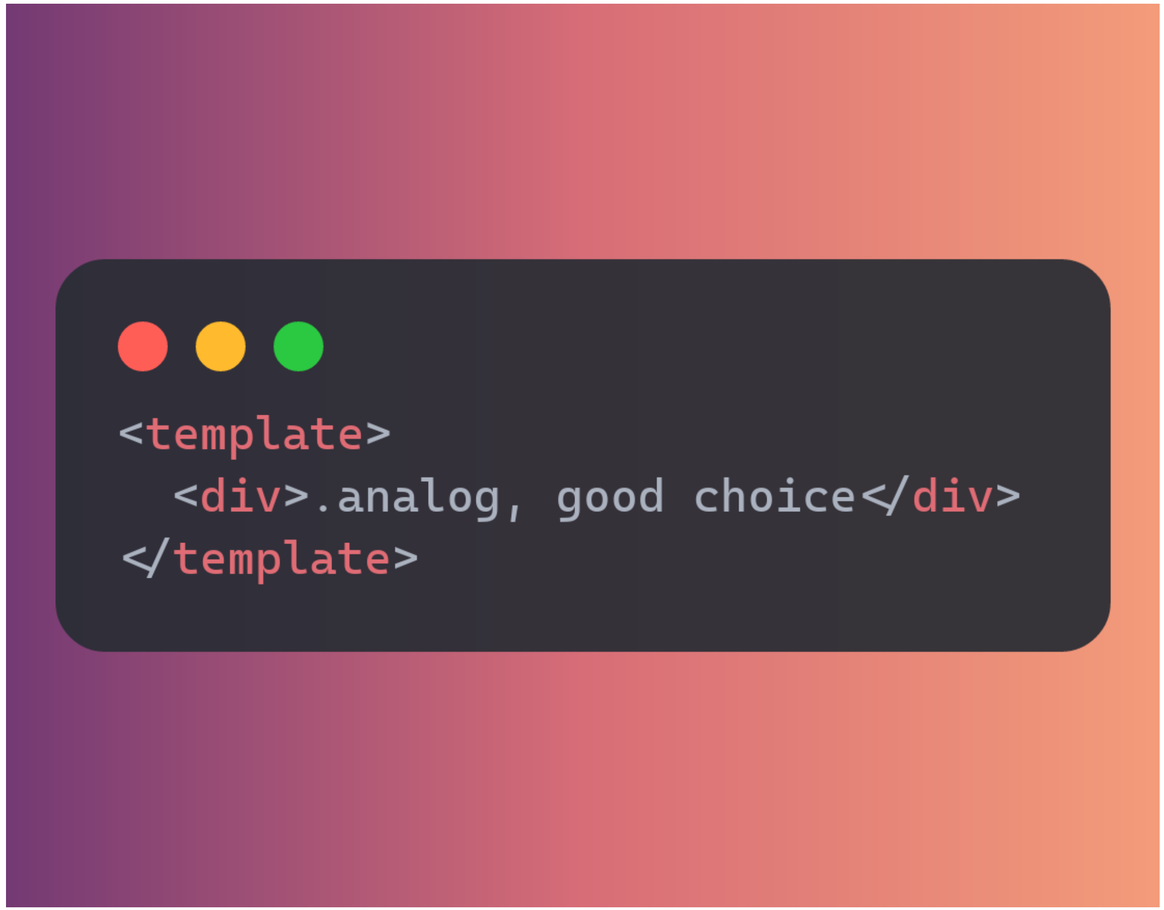
- Currently the most simple Angular SFC would look like
- 7 LOC just to render “angular SFC”

```
import { Component } from "@angular/core";

@Component({
  selector: "app-about",
  template: `<div>angular SFC</div>`,
  standalone: true,
})
export default class AboutComponent {}
```

.analog SFC

NEW WAY

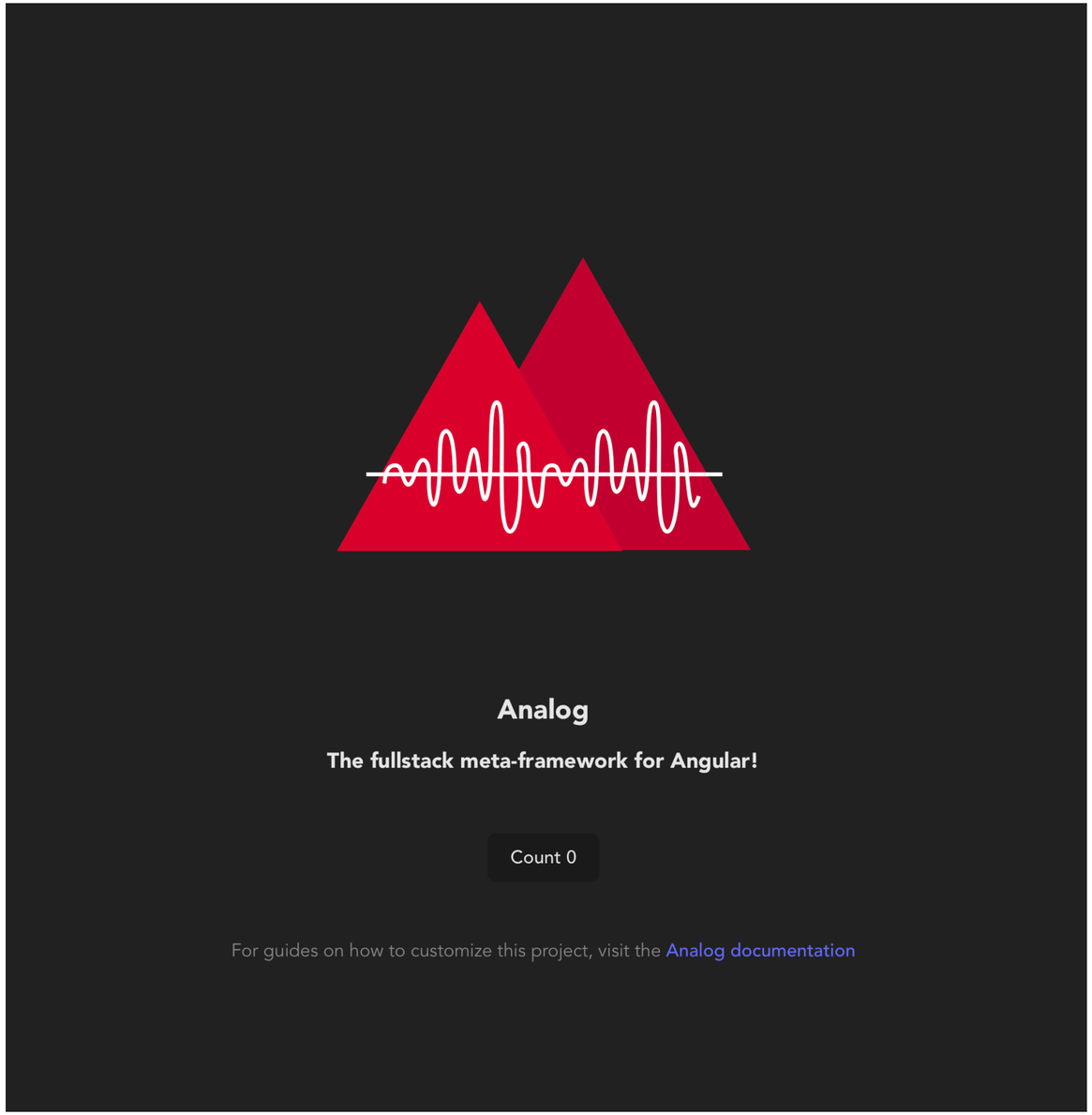
- With .analog SFC a component can be as simple as
- MUCH LESS BOILERPLATE



.analog SFC

NEW WAY

- LETS REVAMP THE OG index.page.ts



.analog SFC

index.page.ts (old way)

```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-home',
  standalone: true,
  template: `
    <div>
      <a href="https://analogjs.org/" target="_blank">
        <img alt="Analog Logo" class="logo analog" src="/analog.svg" />
      </a>
    </div>

    <h2>Analog</h2>

    <h3>The fullstack meta-framework for Angular!</h3>

    <div class="card">
      <button type="button" (click)="increment()">Count {{ count }}</button>
    </div>

    <p class="read-the-docs">
      For guides on how to customize this project, visit the
      <a href="https://analogjs.org" target="_blank">Analog documentation</a>
    </p>
  `,
  styles: [
    `
      .logo {
        will-change: filter;
      }
      .logo:hover {
        filter: drop-shadow(0 0 2em #646cffaa);
      }
      .read-the-docs {
        color: #888;
      }
    `,
  ],
})
export default class HomeComponent {
  count = 0;

  increment() {
    this.count++;
  }
}
```

.analog SFC

index.page.ts (new way)

```
<script lang="ts">
  // counter.analog
  import { signal } from '@angular/core';

  const count = signal(0);

  function increment() {
    count.set(count() + 1);
  }
</script>

<template>
  <div>
    <a href="https://analogjs.org/" target="_blank">
      <img alt="Analog Logo" class="logo analog" src="/analog.svg" />
    </a>
  </div>

  <h2>Analog</h2>

  <h3>The fullstack meta-framework for Angular!</h3>

  <div class="card">
    <button type="button" (click)="increment()">Count {{ count() }}</button>
  </div>

  <p class="read-the-docs">
    For guides on how to customize this project, visit the
    <a href="https://analogjs.org" target="_blank">Analog documentation</a>
  </p>
</template>

<style>
  .logo {
    will-change: filter;
  }
  .logo:hover {
    filter: drop-shadow(0 0 2em #646cffaa);
  }
  .read-the-docs {
    color: #888;
  }
</style>
```

.analog SFC
HOW TO ENABLE?

As mentioned it is experimental so there are a few things we need to do to enable them

.analog SFC

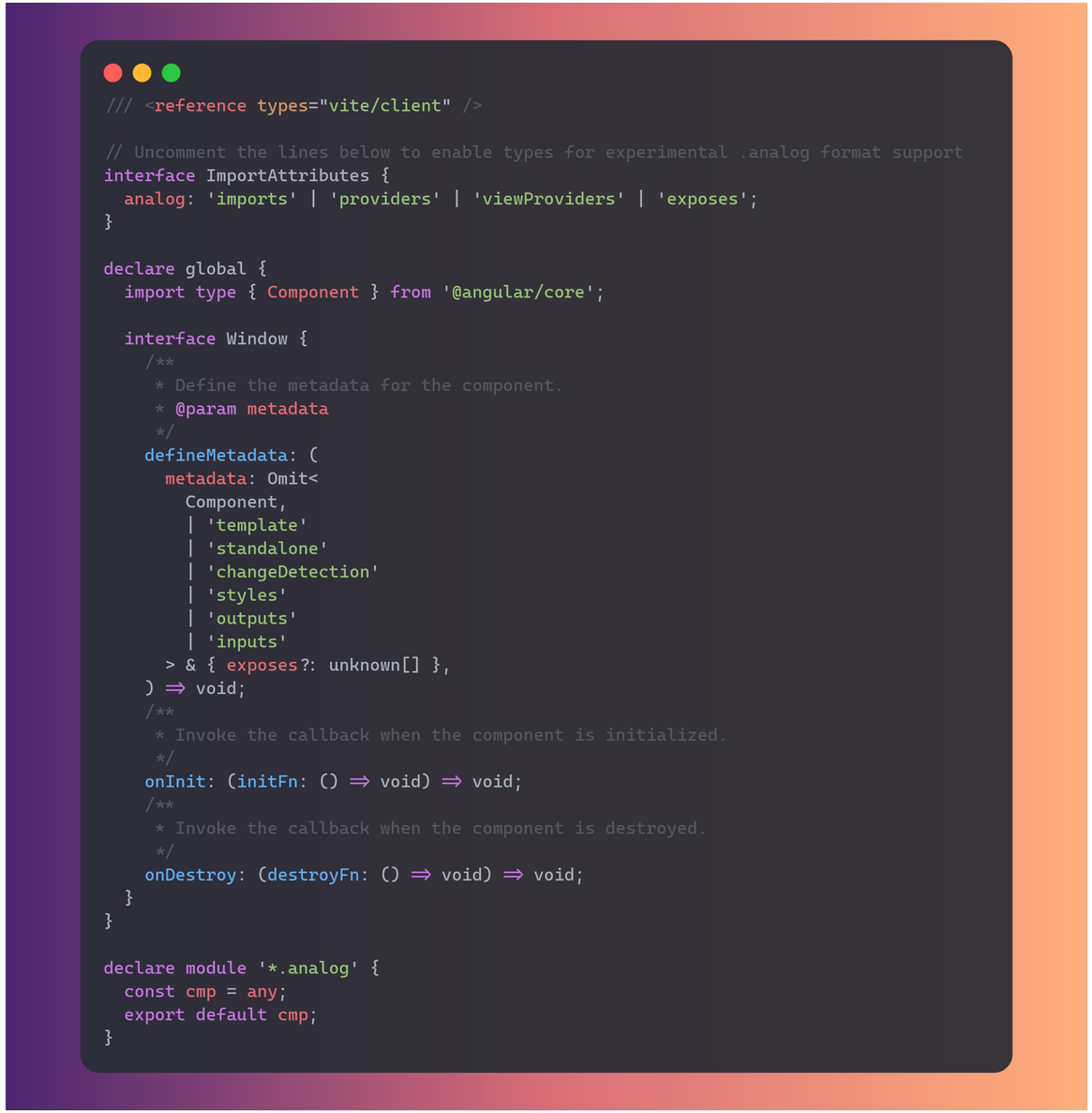
Enabling - Update

vite.config.ts

```ts
// vite.config.ts

import { defineConfig } from "vite";
import analog from "@analogjs/platform";

export default defineConfig({
  // ...
  plugins: [
    analog({
      vite: {
        experimental: {
          supportAnalogFormat: true,
        },
      },
    }),
  ],
});
```

.analog SFC

Enabling - Uncomment the type information in the `src/vite-env.d.ts`



```ts
/// <reference types="vite/client" />

// Uncomment the lines below to enable types for experimental .analog format support
interface ImportAttributes {
  analog: 'imports' | 'providers' | 'viewProviders' | 'exposes';
}

declare global {
  import type { Component } from '@angular/core';

  interface Window {
    /**
     * Define the metadata for the component.
     * @param metadata
     */
    defineMetadata: (
      metadata: Omit<
        Component,
        | 'template'
        | 'standalone'
        | 'changeDetection'
        | 'styles'
        | 'outputs'
        | 'inputs'
      > & { exposes?: unknown[] },
    ) => void;
    /**
     * Invoke the callback when the component is initialized.
     */
    onInit: (initFn: () => void) => void;
    /**
     * Invoke the callback when the component is destroyed.
     */
    onDestroy: (destroyFn: () => void) => void;
  }
}

declare module '*.analog' {
  const cmp = any;
  export default cmp;
}
```

.analog SFC

Enabling - Files outside a project's root

```
vite: {
  experimental: {
    supportAnalogFormat: {
      include: ['src/app/components/auth/**/*'],
    },
  },
},
```

.analog SFC

CAVEATS

- Currently there is a lack of IDE support for VS Code (in progress) and Neovim LSP but there is a webstorm plugin
- You cannot use decorator APIs (@Input, @Component, @ViewChild), must be new function based approach
- You must have lang="ts" present in the <script> tag

.analog SFC

Metadata

- We can modify metadata attributes with `defineMetadata` which works similar to the `@Component` decorator
- Some properties are disallowed such as
  - `template` (use the `<template>` tags or `templateUrl`)
  - `standalone` (always `true`)
  - `changeDetection` (always `OnPush`)
  - `styles` (use `<styles>` tag),
  - `outputs`/`inputs` (Use the signal APIs)

```
defineMetadata({
  host: { class: "block articles-toggle" },
  animations: [toggleAnimation], // animations trigger
  schemas: [CUSTOM_ELEMENTS_SCHEMA], // schemas go here (e.g. web components)
});
```

.analog SFC

Using external template & styles

If you like the authoring experience of Analog SFC but still prefer external template or styles, that can be done with `defineMetadata`

.analog SFC

Importing Components, Services, etc

Imported components need to be added to the appropriate `defineMetadata` property (e.g. `imports` or `providers`) or using `import attributes`

```
import { RouterOutlet } from '@angular/router';
import { AuthService } '../../services/';
import LoginForm from '../../components/auth/login-form.analog';


defineMetadata({
  imports: [RouterOutlet, LoginForm],
  providers: [AuthService]
});
```

.analog SFC

Imports

- When using the Analog format, you do not need to explicitly export anything; the component is the default export of the .analog file

```
<script lang="ts">
  import { output } from '@angular/core'
  const onSubmitForm = output();
</script>

<template>
  <button (click)="onSubmitForm.emit({ username: '', password: '' })">
    login here
  </button>
</template>
```

.analog SFC

Import attributes

- We can avoid using `defineMetadata` by using import attributes

.analog SFC

Inputs and Outputs

- Inputs and outputs are supported using the new Angular signals API
- Can’t use `@Input` and `@Output` decorators

```
<script lang="ts">
  import { inject, signal, DestroyRef } from '@angular/core';
  import { takeUntilDestroyed } from '@angular/core/rxjs-interop'
  import { tap } from 'rxjs';
  import { AuthService } '../../services/' with { analog: 'providers'};
  import LoginForm from '../../components/auth/login-form.analog' with { analog: 'imports' };

  const authService = inject(AuthService);
  const destroyRef = inject(DestroyRef);
  const loading = signal(false);

  const login = (credentials: { username: string; password: string }) => {
    loading.set(true);
    authService.login(credentials).
      pipe(
        takeUntilDestroyed(destroyRef),
        tap(() => loading.set(false)))
      .subscribe()
  }
</script>

<template style="display: block;">
  @if(!loading()) {
    <LoginForm (onSubmitForm)="login($event)" />
  } @else {
    <div>loading</div>
  }
</template>
```

.analog SFC

Lifecycle Methods

- Currently `.analog` SFC supports two Angular Component lifecycle methods, `onInit` and `onDestroy`

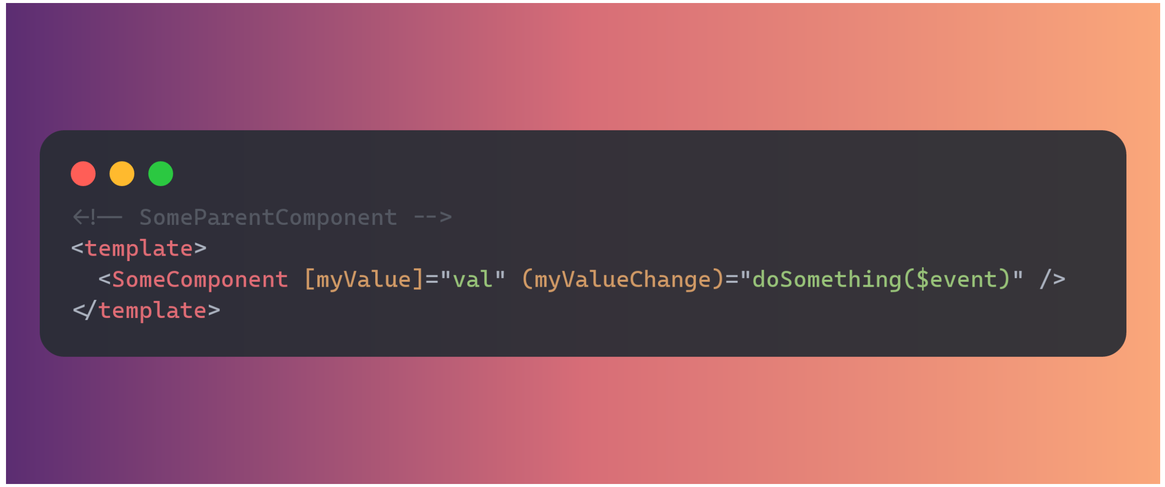
.analog SFC

Models

- Models works just as well
- The input name is taken either from the class member or from the alias option
- The output name is generated by taking the input name and appending Change

.analog SFC

Models



.analog SFC

Directives

- Directives are defined by creating an `.analog` file without a `template` or `templateUrl` passed into `defineMetadata`

```
<script lang="ts">
  import { inject, ElementRef, afterNextRender, effect } from "@angular/core";

  defineMetadata({
    selector: "input[directive]",
  });

  const elRef = inject(ElementRef);

  afterNextRender(() => {
    elRef.nativeElement.focus();
  });

  onInit(() => {
    console.log("init code");
  });

  effect(() => {
    console.log("just some effect");
  });
</script>
```

.analog SFC

Components with Markdown

- Instead of authoring Markdown components with HTML, then can be authored through .analog SFC as well

.analog SFC

Components with Markdown

- And supports “typical” component logic such as creating a `routerLink` or importing other components

```
<script lang="ts">
import { RouterLink } from '@angular/router' with { analog: 'imports'};

const framework = 'Angular';
</script>

<template lang="md">
## About Analog

Analog is a meta-framework for {{ framework }}.

<a routerLink="/contact">Go to contact page (with `routerLink`)</a>

or

[Back Home (with a standard markdown link)](./)
</template>
```