# Design system styling solutions

---

‼️ The examples are just illustrative, do not copy paste them but rather adapt them. Most of them are for the sake of the exploration and are not even examples to copy paste from. These examples rely heavily on the heuristics of the reader, for example

```typescript
// we are going to "import" things from MADE_UP_LIBRARY when they actually do not exist
//do not look for this library in npm
import getPropsFromSlug from "MADE_UP_LIBRARY";

export default async function Component({slug}){
  // we do not provide any details of the getPropsFromSlug
  // and it is neither a real function of some library or a function of the ecosystem/framework
  // is a made up function for the sake of the explanation
  const data = await getPropsFromSlug(slug)
}


```

This means that the reader must already have mastered the essential concepts of **JavaScript, CSS, react (from 17 to 19), browser parsing, rendering strategies, design systems, microfront-end, next JS (from 13 to 15)** or consider reviewing the documentation of those mentioned technologies when needed.

Also, the discussions here are just examples, meaning that if we use a well-known library in a snippet, it DOES NOT MEAN THAT YOU MUST USE THAT LIRBARY AS WELL without considering the tradeoffs.

The MADE_OF_LIBRARY does not mean that you should look for a library that solves the problem, for it could most likely does not exist (other way, we already would have implemented it) or if exists be careful with the security considerations first. It is most likely that you are going to create an implementation of it.

In conclusion, the snippets are not to copy paste in production even local code. Do not expect them to work if you copy paste them.

## 🦏 Question

How to create an performant, flexible and dynamic in some places styling solution for SSG, SSR and CSR

## ✨ Background context

We have had some initial experiments

### With plain CSS

We initially handmade classes like this:

```css
.button{
  display: flex;
  /* some base styles */
  &-primary{
    /*uiTheme
    is custom postcss function we created to use css vars or hardcode the value,
    depends on config.
    */
    background-color: uiTheme(color-button-background-primary-default);

    &:hover, &[data-hover="true"]{
```

```
      background-color: uiTheme(color-button-background-primary-hover);
    }
  }
}

```

And use CVA to resolve the variants

TypeScript

```typescript
import { cva } from 'class-variance-authority';

export const  resolveButtonClassname = cva(["button"],{
  variants: {
    appereance: {
      primary: "button-primary";
    }
  }
})

```

## Trade offs

While this was very straightforward, it led to a lot of CSS duplication; for instance, lets imagine another component that very much use the `display: flex;` attribute. This is not something that is trivial, it creates a lot of repeated code, and we have no control over it.

That is why having utility classes is nice, but then, those need to override everything, are not completely suited to use for every case.

Also here is a potential problem for instance lets imagine we have lots of React components each withs its correspondent CSS file. When importing them (even if when are not using them), the CSS files are going to be considered in the final CSS bundle. One way to avoid that in CSR SPAs is to lazy load the pages and the CSS files COULD be lazy loaded as well, depending on the bundler and the configuration. But with SSG and SSR we have no way to do that: all the components should be registered so the shared CSS files take in consideration all the CSS even if some components are not being used.

Some books argue that GZIP just loves repetition so even if we have a lot of CSS that has duplications, if we activate GZIP, it would lessen seriously the size of the CSS files.

When used with React, if not optimized we can end having lots of executions of CVA, but if we use a memo then the component is not anymore, a potential server component. Take for instance button.

TypeScript

```typescript
import { extractStyleProps , resolveOwnProps } from "MADE_UP_LIBRARY";


export default Button({as, resolveButtonClassname, resolveStyleProp, ...props}){
   const resolvedClassName = resolveButtonClassname(extractStyleProps(props));
   const resolvedStyle = resolveStyleProp(extractStyleProps(props));

   const Component = props.href ? 'a' : as ?? 'button';
   const ownProps = resolveOwnProps(props);

   return  <Component
     {...ownProps}
    style={resolvedStyle}
    className={resolvedClassName}
   />
}


```

If the former code is resolved in the server there is not much problem, since it is going to be plain html at the end. But if we are inside a client component, every time the parent re renders, this button is going to re render as well and recalculate.

## Advantages

This is very reusable like a bootstrap library with any technology, even pure html (very portable)

## Tailwind CSS

Tailwind CSS per se is not attached to any framework. But most of the time it already has some default configuration or plugins to be used with any framework. Inside it essentially works like this:

TypeScript

```typescript
import { compile } from 'tailwindcss'
import getCandidatesFromFiles from "MADE_UP_LIBRARY";

const cssFileContent = readSyncCssFile("somePathWithBaseTailwindConfig.css");

const { build } = await compile(cssFileContent);

// This candidates are the classes that exist from your
// source files, lets imagine ["flex", "flex-col"]
const candidates = await getCandidatesFromFiles()

//It will generate only the classes of the candidates
//for our example would be .flex{display: flex;} .flex-col{flex-direction: column;}
const generatedCsss = build(candidates)

```

Developers tend to confuse the inner process and when they read the documentation that tailwind only generates the classes you need and nothing else it means it reads your source files looking for candidates, for instance if you have an old Button

TypeScript

```typescript
/**
* @deprecated
*/
export default function OldButton(props){
   return <button className="grid" {...props}  />
}


```

Even if never use the OldButton no more, tailwind is going to identify "grid" as a candidate and create the utility class.

The solution some people imagine is to just delete the old button if is not being used any more but lets see another scenario where this continue standing a problem: A CMS powered system.

When we use a CMS to get data and render inside React or other framework, usually the components shoudl be preregistered because you do not know what components are going to be used for each page ahead of time, it will change from the CMS. To illustrate this better lets imagine that the CMS returns something like this:

JSON

```json
{
  "data":[{
    "componentName": "Button",
    "ownProps": {
     "appereance": "primary"
    }
  }
  {
    "componentName": "Accordion",
    "ownProps": {
     "appereance": "primary"
    }

  }
 ]
}

```

Since `componentName` could be any component and change from requests, we need ALL POSSIBLE COMPONENTS TO BE REGISTERED IN THE APP, something like this:

```typescript

import getComponentData from "MADE_UP_LIBRARY";

const components = {
  "Button": Button,
  "Accordion": Accordion,
  "AnotherComponent": AnotherComponent,
  /**More components */
}

export async function ComponentBuilder({slug}){
  const {ownProps, componentName}  = getComponentData(slug)
  const Component = components[componentName]

  return <Component {...props} />
}

```

This only means that tailwind is going to treat all your registered components files as candidate data even if from your CMS never are used all of them and practically you would have all tailwind classes in your shared CSS file.

As a base line let's check for a similar library with lots of utility classes: Bootstrap.

Bootstrap minified CSS weights almost 300kb, which means that with utility classes we might see less than this, so tailwind generated classes probably won't need an optimization right now in the worst scenario that all classes are being generated.

### Trade offs

All classes are generated in build time, nevertheless we can generate them in the server as well via endpoints.

### Advantages

Generates in build time the classes needed (do not think they are being purged in runtime) and those are utility first, so they have a small footprint.

## Some conclusions

It appears that the footprint of tailwind will be less than 300kb in the shared CSS file, so it would be a premature optimization right now to try to JIT compile in the server, in any case let me describe how that optimization would work, for instance in Next.js:

```typescript

import getCriticalCandidatesBySlug from "MADE_UP_LIBRARY";
import { compile } from 'tailwindcss';

export default function Layout({params, children}){
  const { slug } = await params

  const candidates =  await getCriticalCandidatesBySlug(slug)

  const { build } = await compile(tailwindBaseFileData);


  const tailwindCssClasses = build(candidates);

  return <html>
   <meta>
     <style>
       {tailwindCssClasses}
     </style>
  </meta>
    <body>
```

```
      {children}
    </body>
  </html>
}
```

This example is for inlining critical CSS; but let's see another example, first the api route :

TypeScript

```typescript
//styles/[...slug]/router.ts

import getNonCritricalCandidatesBySlug from "MADE_UP_LIBRARY";
import { compile } from "tailwindcss";

export async function GET(_request, {params}){
  const { slug } = await params;

  const candidates = await getNonCritricalCandidatesBySlug(slug);

  const { build } = await compile(tailwindBAseFileData)
  const tailwindCssClasses = build(candidates)

  return new Response(tailwindCssClasses, {
        headers: { 'Content-Type': 'text/css' }
    })

```

This is because we would want to generate the needed CSS from the server and consume it from the client in the following way:

TypeScript

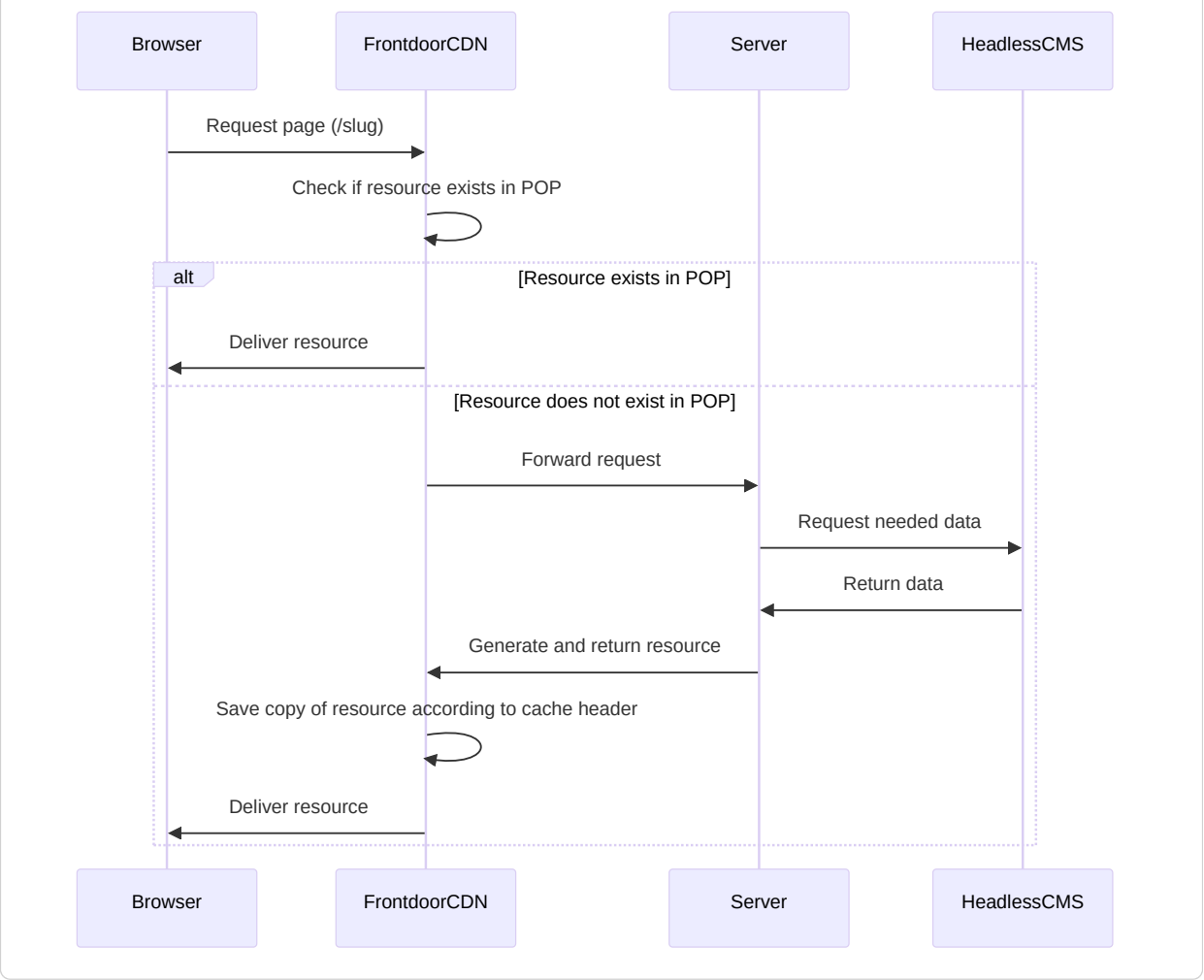
```typescript

export default async Page({params}){
  const { slug } = await params;

  return <div>
    <link rel="styleshett" href={`/styles/${slug}`} />
    {/* some components*/}
  </div>
}

```

But again, this could be a premature optimization. And this would add more time to resolving the css and the HTML from the server, but this is largely mitigated by the fact we are going to use cache in the edge the memorize the responses of CSS, HTML and JS files. We are going to explore how the cache works

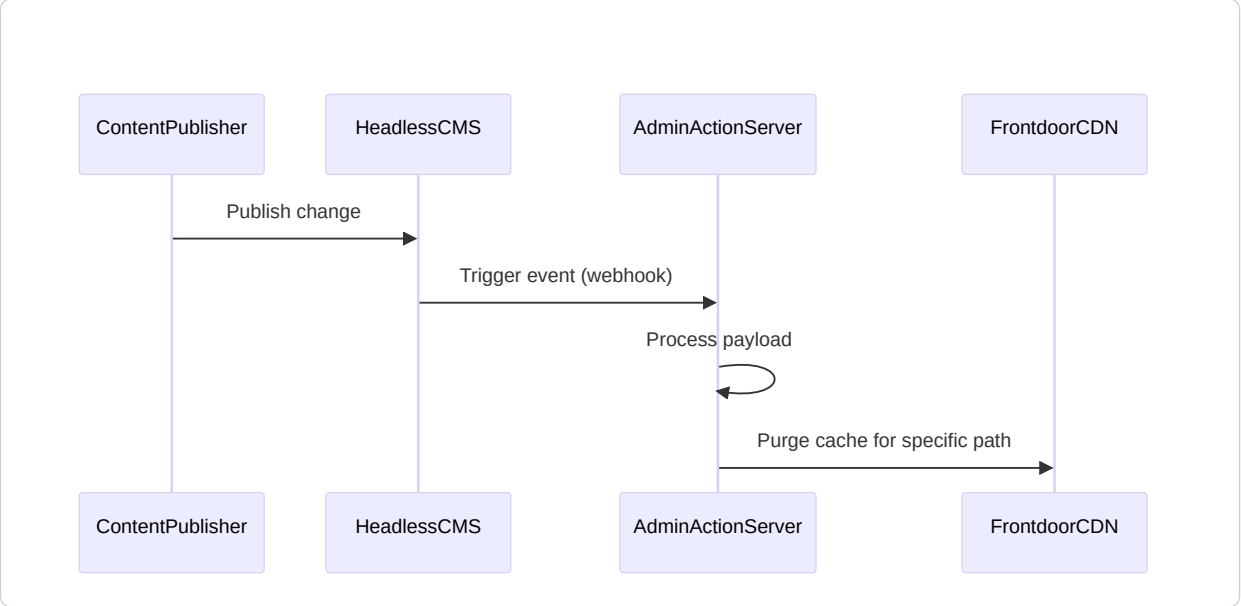
## Cache strategy

POP: Point of persistence

As you can see in the following diagram only if the page is not already in the cache is generated by the Server, avoiding the need to regenerate the page and spend resources (For example, only the first request gets all the down time of creating the page, following requests are cached. If do not want the first user to experience some latency we can even pre warm the cache by scraping the pages, so the first request is made by the scrapper and not a user.)




The you would be asking yourself if the file its cached how the content is going to be refreshed every time, we change the content in the CMS. And this would be a solution.




## Style attribute with CSS vars as a dynamic styling solution

We have the constraint that most of the styles should be applied via class and not the inline style attribute, this is because we end creating a specify war, meaning that since the styles in the style attribute have more specificity than other selectors we might end using !important to override behavior in time of need and since some styles are inherit we would have a hard time reaching what style is causing problems.

For this, we tend to normalize things and keep the specificity at 0,1,0 (only styling via classes).

But we can have another use for the style attribute, we can set there the value of a CSS variable to control the style that is applied via classes. Let's explore a simple example with react.

CSS

```css
/*style.css*/
.custom-grid-item{
  grid-column-end: span var(--number-of-columns, 1);
}

```

TypeScript

```tsx
import "./styles.css";

export default CustomGridItem({numberOfColumns, children}){
  return <div style={{
     "--number-of-columns": numberOfColumns
   }}>
   {children}
   </div>
}


```

When changing CSS vars from the style attribute that change is only applied to the element and its children but does not apply for all the document, meaning that if the value of the CSS var was red by a sibling element, it would have reset its value and not inherited.

But this inheritance represents a problem as well. For instance:

TypeScript

```tsx
<CustomGridItem numberOfColumns={3}>
  <CustomGrid>
    {/* even if did not setted here the number of columns, it will be 3 beacuse
      * the parent CustomGridItem already setted the CSS var value and is inherit by
      * the children. This could lead to unexpected behavior since we were expecting 1
as default
     */}
    <CustomGridItem />
  </CustomGrid>
</CustomGridItem>


```

This could be mitigated by relying in JavaScript to reset the value

TypeScript

```tsx

import "./styles.css";

export default CustomGridItem({numberOfColumns = 1, children}){
  return <div style={{
     "--number-of-columns": numberOfColumns
   }}>
   {children}
   </div>
}


```

Concerns arise when we need to stablish different values for different breakpoints, like the following

TypeScript

```
import "./styles.css";

export default CustomGridItem({numberOfColumns, children}){
  const {
  base,
  fromXSmall = base,
  fromSmall = fromXSmall,
  fromMedium = fromSmall
  } = numberOfColumns

  return <div style={{
     "--number-of-columns": base,
     "--number-of-columns-from-XSmall": fromXSmall,
     "--number-of-columns-from-small":  fromSmall,
     "--number-of-columns-from-medium": fromMedium
   }}>
   {children}
   </div>
}
```

There are some libraries that do the heavy lifting for us but keep in consideration the differences between server generated HTML and client side.

## About styling

Finally, we have concurred (Front end champions) that it is not the responsibility of the components to encapsulate styling but rather to provide either a `className` or a `style` prop solved in the parent, it would look something like this

TypeScript

```
import { tv } from "tailwind-variants";

//remeber that we cannot creat tailwind dinamic classes since
//they are not detected by the tailwind compiler, they have to be
//explicity defined one by one
export const gridItemClassNameResolver = tv({
 variants: {
  numberOfColumns: {
    1: ["col-end-[span 1]"],
    2: ["col-end-[span-2]"],
    3: ["col-end-[span-3]"],
    //14 times (12 column system have two extra columns fo the
    // first and last column actually are "margins" since in some cases
    // desingers want to  overflow the margins and go full width
  }
 }
})
```

And we would end using like this:

TypeScript

```
import { gridItemClassNameResolver } from "MADE_UP_DESING_SYTEM_HELPERS"

expoer default function SomeLayout(){
  return <div className={gridItemClassNameResolver({
    numberOfColumns: 3
  })} />
}
```

This shifts the responsibility no to use a component but rather the resolvers to get the expected behavior. In the case of RSC this is even more flexible (coming from a CMS).

TypeScript

```tsx
export default async function Component({
    as:asProp,
    _componentId,
    componentName,
    ownProps:originalOwnProps
  }){
  // Imagine we would have a Inversion of control container
  // where we decide to use tailwind variants or cva or other solution
  const getPropsByComponentName = resolveFromIOC("getPropsByComponentName")
  // imagine we have in the CMS a content type called Grid,so componentName = "Grid"
   const {
     resolvedClassname,
     style,
     ownProps,
     defaultAsProp,
     childrenComponents
  } = await getPropsByComponentName(componentName, originalOwnProps)

   // custom compoennts would be hash map of compoennts with heavy logic inside but no
   // styling  whatsoever or very well know components that are not polymorphic
   const ResolvedComponent = customComponents[componentName]
     ?? asProp
     ?? defaultAsProp
     ?? 'div';

   const { children, ...restOwnProps  } = ownProps
   return <ResolvedComponent>
     {
       childrenComponents
         ?.map(({componentId, ...props})=><Component key={componentId} {...props}/>)
         ?? children
     }
   </ResolvedComponent>
}


```

This would ensure maximum flexibility, nevertheless, some developers might be struggling with tight deadlines, so they would rather have some ready to use components, layouts, etc. Those could be created as well, like wrappers, because the className + styles resolvers would be fantastic for any framework, that is why we think that the first real design system pattern library would be Json's with tailwind variants alike config, that can be used with tailwind variants., tailwind merge, cva or even in the server to generate CSS ahead of time with custom components by reading this variants ahead of time and creating CSS files with the `@apply` directive to finally have CSS files that can be used in plain html.

## Constraints

- Pages should have little FCP (green, less than 2) using the solution
- The final CSS file should be less than 300kb
- Avoid licking styles the children
- Specificity should be 0,1,0 96% of the time.
- Styling states like hover and active should be repeated when using its corresponding data-* attributes to force them use that state in some contexts like controlled components or at least have a class resolver for force states.
- Only needed classes and styles should be generated per page and shared CSS should be minimum.
- To be possible in build time to decide which classes should be generate as shared and which as per page.
- Critical CSS should be inline
- The unused CSS coverage should not be any of the following: more than 50kb or more than 60% of the file.
- We will not use styled components neither other solution that make the component client side unnecessarily
- Servers should work with no more than 2gb of ram and 2 cores.
- The performance index should be more than 60 in mobile.
- Every project should have a manifest of what CSS is considered critical
- Photos should be loaded asynchronously

- Keep the client components at minimun
- Client components should be able to pass from props Server components instead of creating them and converting them into more Client components.

## 🤔 Assumptions

- Add a list of things that you assume are true/untrue here

## 🏓 Compare ideas

| | Idea | Pros | Cons | Votes |
|---|---|---|---|---|
| 1 | | • | • | +0 |
| 2 | | • | • | +0 |
| 3 | | • | • | +0 |

## 🚀 Next Steps

| | Task | Assigned to | Due date | Bucket |
|---|---|---|---|---|
| 1 | ○ | | | |
| 2 | ○ | | | |
| 3 | | | | |

## 🎉 Final decision

Add final decision here.