

Jo Dikhta Hai, Vo Bikta Hai

Explore all the ways of writing CSS

Using CSS - CSS can be added to HTML documents in 3 ways :

- Inline
- Internal
- External
 - **Inline**- by using the style attribute inside HTML elements.

```
<h1 style="color:blue;">A Blue Heading</h1>
<p style="color:red;">A red paragraph.</p>
```

- **Internal**- by using a `<style>` element in the section.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    body {background-color: powderblue;}
    h1   {color: blue;}
    p    {color: red;}
  </style>
</head>
<body>
  <h1>This is a heading</h1>
  <p>This is a paragraph.</p>
</body>
</html>
```

- **External**- by using a `<link>` element to link to an external CSS file.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <h1>This is a heading</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

```
body {  
  background-color: powderblue;  
}  
  
h1 {  
  color: blue;  
}  
  
p {  
  color: red;  
}
```

1. How do we configure tailwindcss ?

- Configuring Tailwind CSS involves a few simple steps. Tailwind CSS is often configured using a configuration file where you can customize various settings, such as colors, fonts, breakpoints, and more. Here's a step-by-step guide:

💡 Step 1: Create a new project (if not already done)

- Ensure you have a new or existing project where you want to use Tailwind CSS.

💡 Step 2: Install Tailwind CSS

- You can install Tailwind CSS using npm or yarn. Open your terminal or command prompt and navigate to your project's root directory. Run one of the following commands:

Using npm:

```
npm install tailwindcss
```

💡 Step 3: Create a Configuration File

Create a configuration file for Tailwind CSS. You can generate a basic configuration file using the following command:

```
npx tailwindcss init
```

This command creates a `tailwind.config.js` file in your project's root directory.

💡 Step 4: Customize the Configuration (Optional)

Open the generated `tailwind.config.js` file, and you can customize various aspects of Tailwind CSS according to your project's needs. This file includes options for colors, fonts, spacing, breakpoints, and more.

For example, you can customize the colors in the `tailwind.config.js` file like this:

```
module.exports = {
  theme: {
    extend: {
      colors: {
        primary: '#3490dc',
        secondary: '#ffed4a',
        // ...add more custom colors as needed
      },
    },
  },
  // ...other configurations
};
```

💡 Step 5: Create CSS File

Create a CSS file where you will import Tailwind CSS and any additional styles. Typically, this file is named `styles.css` or similar. Import Tailwind CSS using the `@import` directive.

```
/* styles.css */
@import 'tailwindcss/base';
@import 'tailwindcss/components';
@import 'tailwindcss/utilities';
/* Add your custom styles here */
```

💡 Step 6: Build Your Styles

Include your CSS file in your HTML or import it in your JavaScript file if you are using a bundler like Webpack.

💡 Step 7: Use Tailwind CSS Classes in HTML

Now, you can start using Tailwind CSS classes in your HTML files to apply styles.

For example:

```
<div class="bg-primary text-white p-4">
  This is a primary-colored box with white text and padding.
</div>
```

💡 Step 8: Build Your Project

Depending on your setup, you might need to build your project to apply the Tailwind CSS styles. If you're using a bundler like Webpack, make sure to run the appropriate build command.

For example, with npm:

```
npm run build or npm start
```

That's it! We've successfully configured and started using Tailwind CSS in your project. Remember to consult the official documentation for more detailed information and advanced configurations.

2. In `tailwind.config.js`, what does all the keys mean (content, theme, extend, plugins)?

In `tailwind.config.js`, the various keys serve different purposes and allow you to customize and configure different aspects of Tailwind CSS. Here's an overview of what each key typically represents:

💡 1. content Key:

Purpose: Specifies the files that Tailwind CSS should analyze to generate its utility classes. Usage:

```
module.exports = {
  content: [
    './src/**/*.html',
    './src/**/*.js',
    // Add more file paths as needed
  ],
  // ...other configurations
}
```

The content key helps Tailwind CSS identify which files to process and extract utility classes from. It is particularly useful when working with frameworks like React or Vue.

🔒 2. theme Key:

Purpose: Defines the default styles and configurations for various aspects of Tailwind CSS, such as colors, spacing, fonts, and more. Usage:

```
module.exports = {
  theme: {
    extend: {
      colors: {
        primary: '#3490dc',
        secondary: '#ffed4a',
        // ...add more custom colors
      },
    },
    // ...other theme configurations
  },
  // ...other configurations
};
```

The theme key allows you to customize default styles and extend or override the default configuration provided by Tailwind CSS. It is where you can define your project-specific design system.

🔑 3. extend Key:

Purpose: Extends or overrides the default configuration provided by Tailwind CSS. Usage:

```
module.exports = {
  extend: {
    colors: {
      primary: '#3490dc',
      secondary: '#ffed4a',
      // ...add more custom colors
    },
    // ...other extensions
  },
  // ...other configurations
};
```

The extend key is often used to add new styles or extend existing ones. It is especially useful for adding project-specific utility classes or modifying existing ones.

🔑 4. plugins Key:

Purpose: Allows you to use or define custom plugins to extend or modify Tailwind CSS functionality. Usage:

```
module.exports = {
  plugins: [
    require('@tailwindcss/forms'), // Example plugin
    // ...add more plugins as needed
  ],
  // ...other configurations
};
```

The plugins key lets you incorporate third-party plugins or create your own custom plugins. Plugins can add new features, styles, or utilities to Tailwind CSS.

These keys provide a flexible and powerful way to configure Tailwind CSS based on your project's requirements. They allow you to control the content, define styles, extend default configurations, and enhance functionality through plugins. Remember to consult the official Tailwind CSS documentation for detailed information on each configuration option and best practices.

3. Why do we have **.postcssrc** file?

The **.postcssrc** file, often named `postcss.config.js`, is a configuration file for PostCSS. PostCSS is a tool for transforming styles with JavaScript plugins, and it is commonly used in conjunction with build tools like webpack or parcel for processing and optimizing CSS.

Here are the primary reasons why you might have a `.postcssrc` file:

- **Plugin Configuration:** The main purpose of the `.postcssrc` file is to configure the plugins that PostCSS should use during the CSS transformation process. These plugins can handle tasks such as autoprefixing, minification, and syntax enhancements.
- **Custom Configuration:** You may need a `.postcssrc` file if you want to customize the behavior of PostCSS beyond the default settings provided by the build tool (e.g., webpack). This allows you to have fine-grained control over the PostCSS transformations.
- **Presets and Options:** PostCSS plugins often come with various options and presets that you can configure based on your project's needs. The `.postcssrc` file is a convenient place to define these options and presets.
- **Maintainability:** Separating the PostCSS configuration into its own file makes the build configuration more maintainable and organized. It allows you to centralize PostCSS-related settings and keep them distinct from other build tool configurations.
- **Sharing Configurations:** Having a dedicated configuration file makes it easier to share and reuse PostCSS configurations across different projects. It can be particularly useful in larger development ecosystems where consistent styles and build processes are desired.

Example `.postcssrc` file:

```
// postcss.config.js
module.exports = {
  plugins: {
    // Example plugins with options
    'autoprefixer': {},
    'postcss-preset-env': {
      stage: 3,
      features: {
        'nesting-rules': true,
      },
    },
    'cssnano': {
      preset: 'default',
    },
  },
};
```

In this example, the `.postcssrc` file configures three PostCSS plugins: `autoprefixer` for adding vendor prefixes, `postcss-preset-env` for enabling future CSS features, and `cssnano` for minification. The options provided for each plugin customize their behavior.

Remember that the specific configuration options and plugins you include in your `.postcssrc` file will depend on your project's requirements and the PostCSS features we want to leverage.

Notes :

- We are using parcel, So, parcel will use `.postCSSrc`.

- `.postCSSrc` is used to understand tailwind.

Example : Configuration file for `PostCSS`

```
{
  "plugins": {
    "tailwindcss": {}
  }
}
```

- So, basically in this configuration, we are just telling that we are using tailwind in our project and please parcel, understand tailwind, wherever you find tailwind classes, understand it. That's all is the way, what we are doing in this file.
- Tailwind css gives you class names automatically for every css that you would want to write into your app.
- Install `Tailwind css intelliSense` extension in vscode to get the hint for using Tailwind.
- If tailwind css extension does not giving suggestion then use `ctrl + spacebar` for mac, & for windows it will be similar.

Pros and Cons of tailwind css

Pros :

- No need to move from js to html to css for changing.

Cons :

- we have to write css so long.