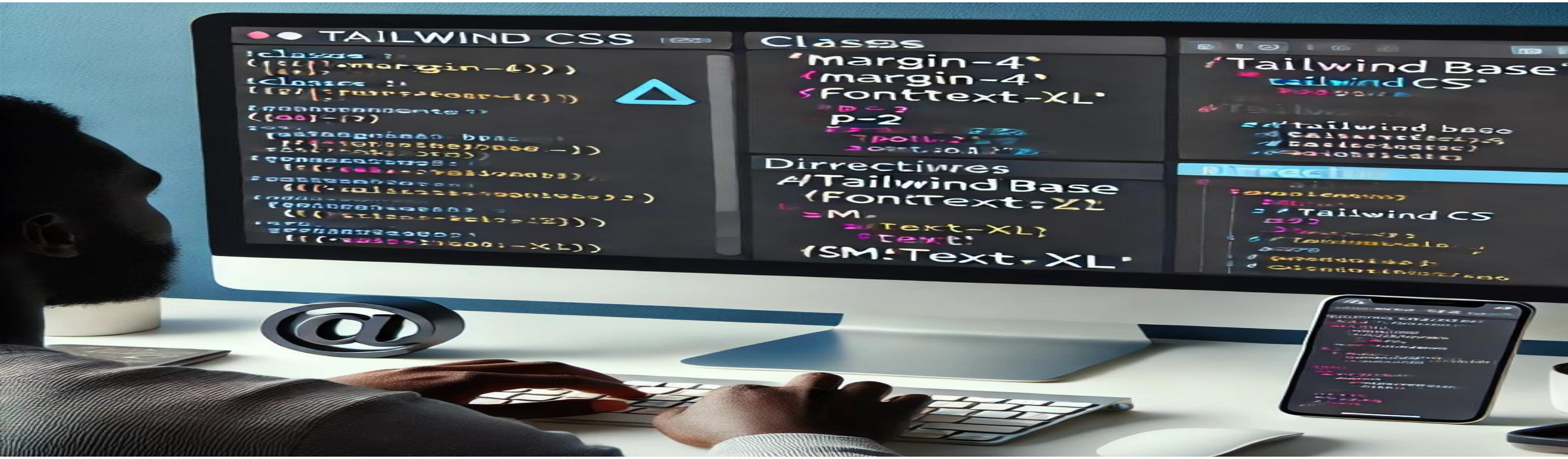
2: Apply Tailwind CSS Framework.



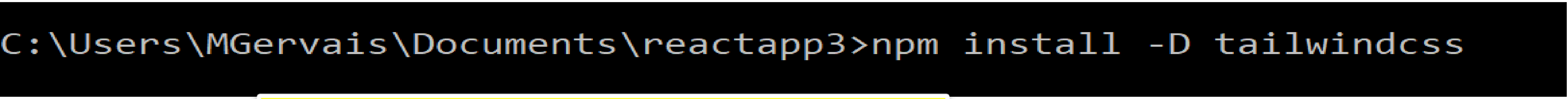
Indicative contents

2.1 Applying Tailwind Utility Classes
2.2 Applying Responsive Design Principles
2.3 Customization of Tailwind Styles

2.1.1: Integrating Tailwind CSS with React.JS

Step 1: Install Tailwind CSS

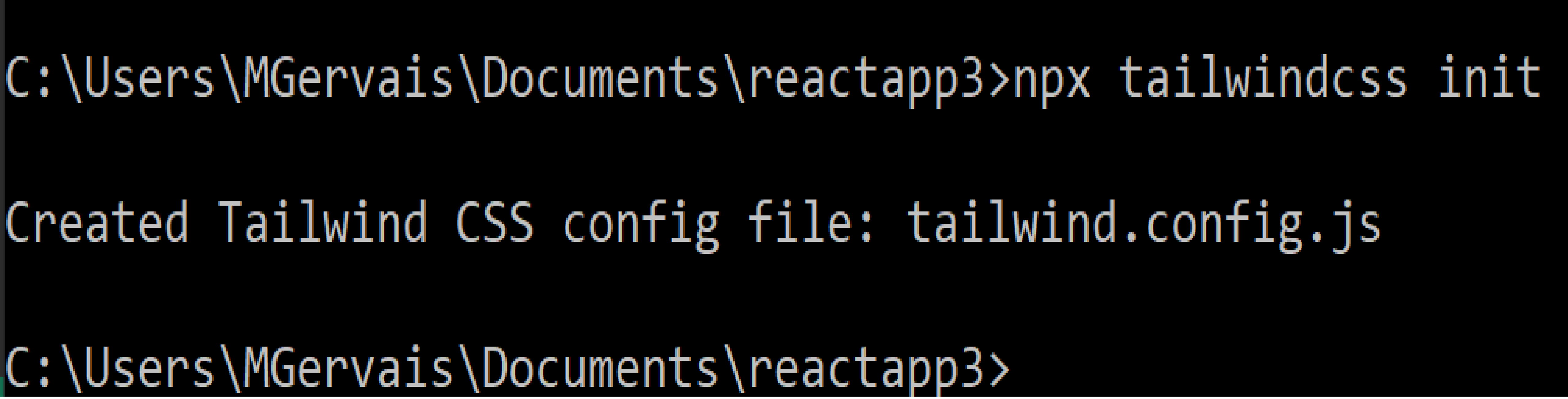
Install tailwindcss via npm by typing this command npm install -D tailwindcss



```
C:\Users\MGervais\Documents\reactapp3>npm install -D tailwindcss
```

Step 2: Create Tailwind.config.js file

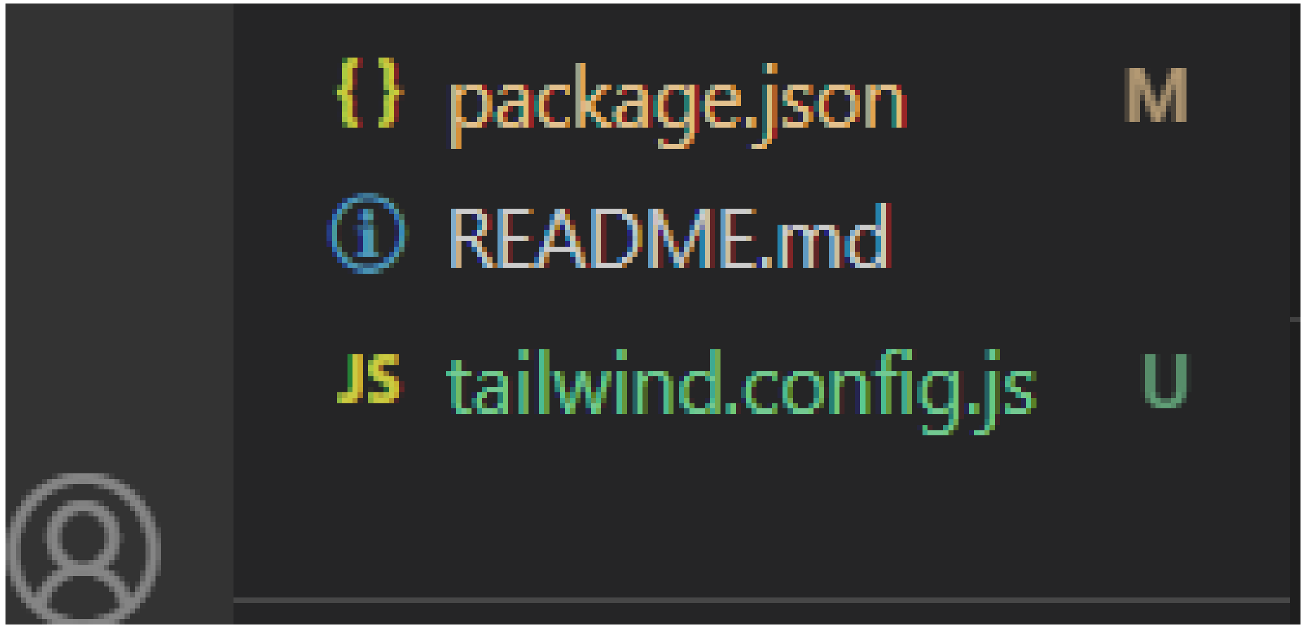
Create your tailwind.config.js file with this command npx tailwindcss init in prompt command.



```
C:\Users\MGervais\Documents\reactapp3>npx tailwindcss init

Created Tailwind CSS config file: tailwind.config.js

C:\Users\MGervais\Documents\reactapp3>
```



```
{} package.json        M
ⓘ README.md
JS tailwind.config.js  U
```

Step 4: Include Tailwind in your CSS

In your React app, locate or create a src/index.css file, and include the Tailwind CSS directives to inject its base styles, components, and utilities.

Replace the contents of src/index.css with:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Step 5: Import the Tailwind CSS in your React application

You need to import the index.css file into your main src/index.js file to ensure Tailwind CSS is applied globally in your React app.

In src/index.js, add the following line: import './index.css';

Step 6:Finally, start your React application: npm start

What is tailwind css

Tailwind CSS is a **utility-first CSS framework** that provides low-level utility classes to build custom designs directly in your HTML, instead of pre-designed components.

Key Characteristics:

1. Utility-First Approach

Instead of predefined components (like Bootstrap's buttons or cards), Tailwind provides single-purpose utility classes

Example:

```
!-- Instead of a "btn" class -->
<button class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4
rounded">
  Button
</button>
```

2. Highly Customizable

Configure everything via **tailwind.config.js**: colors, spacing, fonts, breakpoints, etc.
Can extend or override defaults to match your design system.

3. Responsive by Default

Built-in responsive prefixes:

```
<div class="text-sm md:text-base lg:text-lg">
  Responsive text
</div>
```

4. Component-Friendly

Encourages extracting repeated utilities into component classes (using @apply or JavaScript components)
Works well with React, Vue, Angular, etc.

Example Comparison:

Traditional CSS:

```
.btn {
  background-color: #3b82f6;
  padding: 0.5rem 1rem;
  border-radius: 0.25rem;
  color: white;
}
```

Tailwind Approach:

```
<button class="bg-blue-500 px-4 py-2 rounded text-white">
  Button
</button>
```

Pros:

- ✓ **Rapid development** - No context switching between HTML and CSS
- ✓ **Consistent sizing** - Uses predefined scale (4px base)
- ✓ **Small bundle size** - PurgeCSS removes unused classes
- ✓ **Design consistency** - Enforces design system constraints
- ✓ **Highly customizable** - Not opinionated about design

Cons:

- ✗ Verbose HTML - Can get cluttered with many classes
- ✗ Learning curve - Need to memorize many class names
- ✗ Not for beginners - Requires understanding of CSS concepts

Typical Workflow:

- ✓ Install via npm: npm install -D tailwindcss
- ✓ Generate config: npx tailwindcss init
- ✓ Include in CSS: @tailwind base; @tailwind components; @tailwind utilities;
- ✓ Build with PostCSS

- Custom design systems
- Rapid prototyping
- Projects where design consistency is important
- Component-based frameworks (React, Vue, etc.)

Use Cases:

- Custom design systems
- Rapid prototyping
- Projects where design consistency is important
- Component-based frameworks (React, Vue, etc.)

Tailwind essentially treats CSS as a utility toolkit rather than a semantic styling language, making it popular for developers who want design control without writing custom CSS for every element.

Here are 4 common CSS rules with their Tailwind CSS counterparts:

1. Padding

css

```css
.element {
  padding: 1rem 2rem;
}
```

html

```html
<div class="px-8 py-4">
  <!-- px-8 = 2rem (32px), py-4 = 1rem (16px) -->
</div>
```

CSS:

```
css
.container {
  display: flex;
  justify-content: space-between;
  align-items: center;
  gap: 1rem;
}
```

Tailwind:

```
html
<div class="flex justify-between items-center gap-4">
  <!-- gap-4 = 1rem -->
</div>
```

3. Responsive Typography

```
css

.title {
  font-size: 1.5rem;  /* 24px */
  font-weight: 700;
  color: #1f2937;
}

@media (min-width: 768px) {
  .title {
    font-size: 2.25rem;  /* 36px */
  }
}
```

html

```html
<h1 class="text-2xl md:text-4xl font-bold text-gray-800">
  <!--
    text-2xl = 24px (1.5rem)
    md:text-4xl = 36px (2.25rem) at 768px+
  -->
  Title
</h1>
```

4. Card Component

```css
.card {
  background-color: white;
  border-radius: 0.5rem;
  box-shadow: 0 4px 6px -1px rgb(0 0 0 / 0.1);
  padding: 1.5rem;
  max-width: 28rem;
  margin: 0 auto;
}

.card:hover {
  box-shadow: 0 20px 25px -5px rgb(0 0 0 / 0.1);
}
```

```
<div class="bg-white rounded-xl shadow-lg p-6 max-w-md mx-auto hover:shadow-2xl">
  <!--
    rounded-xl = 0.75rem (12px)
    shadow-lg = 0 10px 15px -3px rgb(0 0 0 / 0.1)
    p-6 = 1.5rem (24px)
    max-w-md = 28rem (448px)
    mx-auto = margin: 0 auto
  -->
  Card content
</div>
```

@apply directive

For reusable component patterns, Tailwind offers @apply:

CSS with @apply:

The @apply directive lets you extract repeated utility patterns while keeping the benefits of Tailwind's design system.

Tailwind CSS Class Breakdown:

```
<h1 class="text-2xl md:text-4xl font-bold text-gray-800">
```

This single line combines 4 different CSS rules with responsive behavior. Here's what each part does:

1. text-2xl

Base font size: 1.5rem (24px)
Applies: font-size: 1.5rem;
This is the default size for all screen sizes (mobile-first)

2. md:text-4xl

Responsive modifier: md: means "medium screens and up" (≥768px)

Font size at medium screens: 2.25rem (36px)

Applies: @media (min-width: 768px) { font-size: 2.25rem; }

Overrides text-2xl on medium screens and larger

3. font-bold

Font weight: 700 (bold)

Applies: font-weight: 700;

Consistent across all screen sizes

4. text-gray-800

Text color: A dark gray from Tailwind's color palette

Hex value: #1f2937 or rgb(31, 41, 55)

Applies: color: #1f2937;

Equivalent CSS:

```
h1 {
  font-size: 1.5rem;       /* 24px - text-2xl */
  font-weight: 700;        /* font-bold */
  color: #1f2937;          /* text-gray-800 */
}

/* Medium screens and up (≥768px) */
@media (min-width: 768px) {
  h1 {
    font-size: 2.25rem;   /* 36px - md:text-4xl */
  }
}
```

Visual Result:

On mobile: 24px bold dark gray text

On tablets/desktops (768px+): 36px bold dark gray text

Tailwind's Breakpoints:

Tailwind has 5 default breakpoints:

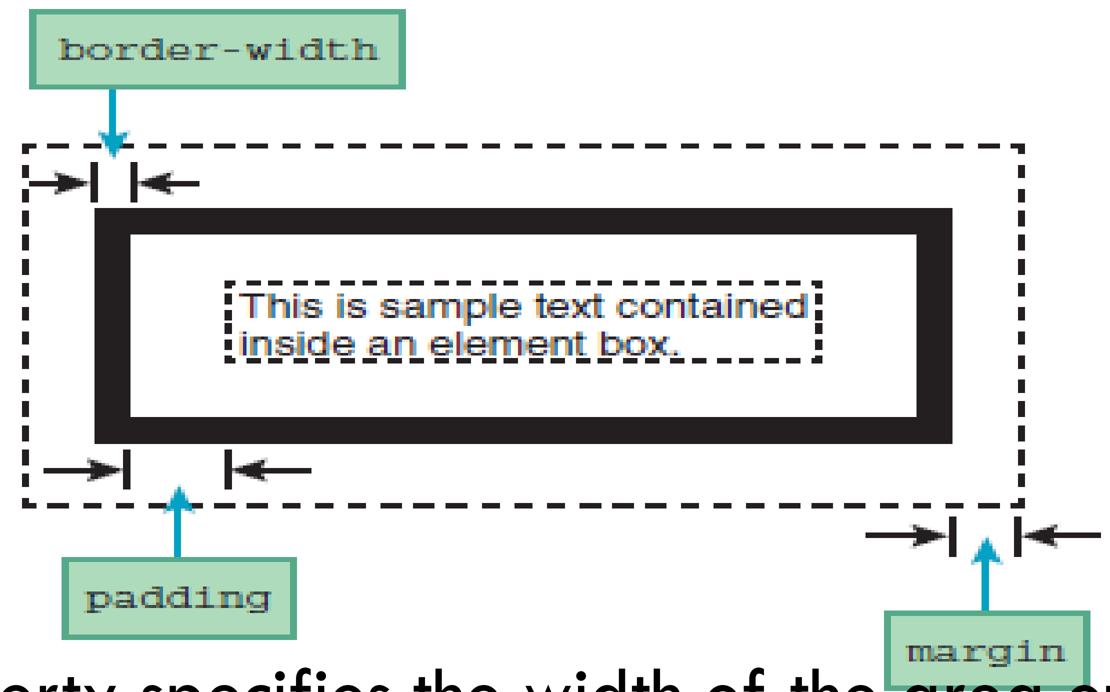
sm: 640px	(small)
md: 768px	(medium)
lg: 1024px	(large)
xl: 1280px	(extra large)
2xl: 1536px	(2x large)

The md: prefix means "apply this style starting from 768px screen width and up."

Why This Approach is Useful:

- **Mobile-first:** Start with mobile styles, enhance for larger screens
- **Atomic:** Each class does one specific thing
- **Consistent:** Uses predefined scale (text-2xl = 24px, text-4xl = 36px)
- **Readable:** The class names explain what they do.

This pattern is extremely common in Tailwind for creating responsive typography that adapts to different screen sizes while maintaining design consistency.



The padding property specifies the width of the area on the interior of an element's border, whereas the margin property specifies the width of the area on the exterior of an element's border.

Usually, the most appropriate type of value for the padding and margin properties is a CSS pixel value. Here's an example CSS rule that uses padding and margin properties: .label {border: solid; padding: 20px; margin: 20px;}

The **margin** and **padding** properties allow negative values. While a positive value forces two elements to be separated by a specified amount, a negative value causes two elements to overlap by a specified amount.

```
.label {
padding: 20px;
margin: 20px;
display: inline-block;
}
```

CSS Font Size Measurement

Absolute Units (Fixed Sizes)

1. Pixels (px)

Most common unit - 1 pixel equals one dot on the screen

```
p {
  font-size: 16px; /* Fixed size, not scalable by user */
}
```

2. Points (pt)

Traditional print measurement (1pt = 1/72 inch)

```
h1 {
  font-size: 24pt; /* Good for print stylesheets */
}
```

3. Inches (in), Centimeters (cm), Millimeters (mm)

Physical measurement units

```
.print-heading {
  font-size: 0.5in; /* Half inch */
  font-size: 1.27cm; /* Same in centimeters */
  font-size: 12.7mm; /* Same in millimeters */
}
```

Relative Units (Scalable Sizes)

4. Ems (em)

Relative to parent element's font size

```
.container {
  font-size: 16px;
}
.container p {
  font-size: 1.5em; /* 24px (16 × 1.5) */}
```

5. Rems (rem)

Relative to root (html) element's font size - avoids compounding

```
html {
  font-size: 16px; /* Base size */}
h1 {
  font-size: 2rem; /* 32px(16 × 2) */
}
p {
  font-size: 1rem; /* 16px (16 × 1) */
}
.small {
  font-size: 0.875rem; /* 14px (16 × 0.875) */
}
```

6. Percentages (%)

Similar to ems - relative to parent element

```
body {
  font-size: 16px;
}

h2 {
  font-size: 150%; /* 24px (16 × 1.5) */
}
```

7. Viewport Units

Relative to viewport dimensions

```
/* Relative to viewport width */
h1 {
  font-size: 5vw; /* 5% of viewport width */
}
/* Relative to viewport height */
h2 {
  font-size: 10vh; /* 10% of viewport height */
}
/* Minimum of vw and vh */
h3 {
  font-size: 5vmin; /* 5% of smaller dimension */
}
/* Maximum of vw and vh */
h4 {
  font-size: 5vmax; /* 5% of larger dimension */ }
```

Keyword Values

8. Absolute Keywords

Predefined sizes

```
p {
  font-size: xx-small;  /* ~9px (assuming 16px base) */
  font-size: x-small;   /* ~10px */
  font-size: small;     /* ~13px */
  font-size: medium;    /* ~16px (default) */
  font-size: large;     /* ~18px */
  font-size: x-large;   /* ~24px */
  font-size: xx-large;  /* ~32px */
}
```

9. Relative Keywords

```
.smaller {
  font-size: smaller; /* ~80% of parent */
}

.larger {
  font-size: larger;  /* ~120% of parent */
}
```

2.1.2: Description of utility first fundamentals and states

Utility-First Fundamentals

Utility-first fundamentals in Tailwind CSS is an approach where you build designs using pre-defined utility classes that are designed to be small, single-purpose, and composable. Instead of writing custom CSS for each component, you apply these utility classes directly in your HTML to style elements. This results in a faster development process and more maintainable code since you can focus on composing utilities rather than writing new styles for each element.
Utility classes are low-level, single-purpose classes that do one thing, such as setting a specific margin, padding, font size, color, etc.

Some features of utility-first fundamentals in Tailwind CSS are:

Single-purpose Classes: Each utility class serves one specific function. For example:

Padding (p-*): Controls padding for an element.

p-4: Adds padding of 1rem.

px-4: Adds padding of 1rem on the left and right (x-axis).

py-4: Adds padding of 1rem on the top and bottom (y-axis).

Margin (m-*): Controls margin for an element.

- m-4: Adds margin of 1rem.
- mx-auto: Centers an element horizontally by setting left and right margins to auto.

Example: <div class="m-4 p-4">Content with margin and padding</div>

Text Size:

- text-sm: Sets the font size to small.
- text-lg: Sets the font size to large.
- text-4xl: Sets the font size to extra-large (font-size: 2.25rem).

Font Weight:

- font-thin: Sets a thin font weight.
- font-bold: Sets a bold font weight.

Text Alignment:

- text-left, text-center, text-right: Aligns text accordingly.

Example: <h1 class="text-4xl font-bold text-center">Centered Heading</h1>

Text Color:

- text-gray-700: Sets text color to dark gray.
- text-blue-500: Sets text color to blue.

Example: <p class="text-gray-700">This is gray text</p>

Background Color (bg-*):

- **bg-white:** Sets the background color to white.
- **bg-blue-500**: Sets the background color to blue.

Example: <div class="bg-blue-500 p-4">This is a blue background</div>

Composability classes: They combine multiple utility classes to achieve complex designs. Instead of creating new CSS rules, you use a combination of classes.

Example:

```
<div class="p-4 bg-blue-500 text-white rounded-lg">
This is a box with padding, background color, and rounded corners.
</div>
```

Responsiveness Built-in classes: Tailwind makes it easy to create responsive designs by providing utilities for different screen sizes. You don't need to write custom media queries.

Example:

```
<div class="p-4 sm:p-6 md:p-8 lg:p-10">
This element has different padding on small, medium, and large screens.
</div>
```

States in Tailwind CSS

Tailwind CSS provides utilities for managing various states, like hover, focus, active, and more. These state utilities are applied by prefixing the utility class with the state modifier.

Hover State: You can define styles that apply when an element is hovered using the hover: prefix.

Example:`<button class="bg-gray-500 hover:bg-blue-500">Hover Me</button>`

Focus State: Styles can be applied when an element is focused with the focus: prefix.

Example: `<input class="border-gray-300 focus:border-blue-500" />`

Active State: The active: prefix allows you to style elements when they are active (being clicked).

Example: <button class="bg-gray-500 active:bg-blue-500">Click Me</button>

Other States:

disabled: for elements that are disabled.
group-hover: for applying hover styles to an element when its parent is hovered.
focus-within: for styling an element when any child element is focused.

2.1.3: Applying Utility First Fundamentals and states

Step 1: Browse to the folder where your project is created
Open File Explorer (Windows) and navigate to the folder where your React project is located.

Step 2: Open the command prompt

In Windows, hold down Shift + Right-click inside the folder, then select "Open PowerShell window here" or "Open Command Prompt here."

Step 3: Type the command to open the project in your text editor

If you are using VS Code as your text editor, you can use the following command:
code.
This will open the current directory in VS Code.

Step 4: Access the React file you want to style

In VS Code, browse through the src folder and open the component file you want to style, typically something like App.js, index.js, or any component file.

Step 5: Use the utility-first fundamentals

If you have Tailwind CSS set up in your project, you can start applying Tailwind utility classes directly in your JSX code.

For example:

```
<div className="bg-blue-500 text-white p-6">
<h1 className="text-2xl font-bold">Hello World</h1>
</div>
```

Step 6: Apply the states

Tailwind makes it easy to style elements based on different states such as hover, focus, active, etc. For example:

```
<button className="bg-gray-500 hover:bg-blue-500 active:bg-blue-700 text-
white font-bold py-2 px-4 rounded">
Click Me
</button>
```

2.1.4: Description of animation and transitions

Tasks

1: You are requested to answer the following questions

i. What do you understand by the term?

a) Animation

b) Transition

ii. Outline four utility classes that are used to animate object with Tailwind CSS.

iii. Enumerate the components of transitions in Tailwind CSS.

iv. Give the main possible classes of:

a) Animation

b) Transition delay

c) Transition timing function

d) Transition duration

e) Transition property

2: Write your findings on papers, flip chart or chalkboard

3: Present the findings answers to the whole class or trainer

4: For more clarification, read the key readings 2.1.4.

2.1.4: Description of animation and transitions

Animation

Animation is the ability to apply motion effects to elements, either using pre-defined utility classes or custom animations. These effects can include transformations like spinning, pulsing, bouncing, or fading, which help create a more interactive and engaging user experience.

Animations in Tailwind are handled with utility classes that apply keyframe-based animations. Tailwind includes predefined animations like bounce, spin, and ping.

Common Animation Classes:

animate-none: Disables animations.

animate-spin: Rotates the element (spinning effect).

animate-ping: Creates a pulsing effect, like a radar ping.

animate-pulse: Causes an element to smoothly increase and decrease in opacity.

animate-bounce: Bounces the element up and down.

Example: Spinning Loader

```
<div class="animate-spin h-8 w-8 border-4 border-blue-500 border-t-transparent
rounded-full"></div>
```

Example: Bouncing Button

```
<button class="bg-green-500 text-white px-4 py-2 rounded-full animate-bounce">
Bouncing Button
</button>
```

Here, the button will bounce up and down continuously because of the animate-bounce class.

Transitions

Transition is the smooth change of CSS properties over a specified duration when an element's state changes, such as when it is hovered, focused, or clicked.

Transitions components in Tailwind CSS are: transition, transition –property, duration-*,ease-*,and delay-*.

Transition Property: utilities for controlling which CSS properties transition.

- **transition-none**: Disables all transitions for an element.

Example: **<button class="transition-none">No Transition</button>**

- **transition-all:** Applies transitions to all properties of an element, such as background color, opacity, transform, shadow, etc.Useful when you want to apply transitions to multiple properties at once.

Example: **<button class="transition-all duration-300">Transition All</button>**

- **Transition:** This is a shorthand class for transition-all. It enables a smooth transition for all properties but without specifying the details. It offers a quick way to apply transitions to all properties without specific customization.

Example: **<button class="transition">Default Transition</button>**

- **transition-colors:** Applies transitions specifically to color-related properties, such as background-color, border-color, and text-color.

Example: **<button class="transition-colors duration-500 hover:bg-blue500">Transition Colors</button>**

- **transition-opacity**:Adds transitions to changes in opacity. Useful for fading effects where you want to control how an element appears or disappears.

Example: <div class="transition-opacity duration-300 opacity-0 hover:opacity-100">Fade In</div>

- **transition-shadow**: Adds transitions to changes in the box shadow. Useful for hover effects where you want to smoothly change the shadow of an element.

Example: <div class="transition-shadow duration-300 hover:shadow-lg">Shadow Transition</div>

- **transition-transform**:Adds transitions to CSS transforms, such as translate, rotate, scale, or skew. Useful when you want smooth animations for transformations like rotating or scaling an element.

Example: <div class="transition-transform duration-300 hover:scale-110">Scale on Hover</div>

Use the transition-* utilities to specify which properties should transition when they change.

Example:

```
<div>
<button className="bg-blue-500 text-white font-bold py-2 px-4 rounded
transition-all duration-300 ease-in-out
hover:bg-blue-700 hover:scale-110"> Hover Me
</button>
</div>
```

Transition duration is the amount of time a transition takes to complete. It's used to control how quickly an element transitions from one state to another, such as when a hover effect changes the color, size, or position of an element. It is defined in milliseconds.

The common transition duration classes:

- duration-75: 75 milliseconds
- duration-100: 100 milliseconds
- duration-150: 150 milliseconds (default)
- duration-200: 200 milliseconds
- duration-300: 300 milliseconds
- duration-500: 500 milliseconds
- duration-700: 700 milliseconds
- duration-1000: 1000 milliseconds (1 second)

Use the **duration-*** utilities to control an element’s transition-duration.

Transition timing function controls how intermediate states of a transition are calculated, defining the speed curve of the transition.

Tailwind provides several built-in timing function utilities, including:

- **ease-linear:** Transitions at a constant speed.
- **ease-in:** Starts slowly and speeds up.
- **ease-out:** Starts fast and slows down.
- **ease-in-out:** Starts slowly, speeds up, then slows down at the end.

Use the ease-* utilities to control an element’s easing curve.

Example:

```
<div class="transition ease-in-out duration-500">
<!-- Content that will transition -->
</div>
```

Transition delay is the time before the transition effect starts when a state change occurs, like hover or focus. It controls how long the system waits before applying the transition effects like changing color, opacity, or transforming an element.

In Tailwind CSS, you can add transition delays using the delay-{time} utility. This defines how long (in milliseconds) a transition should wait before starting. For example, you can use delay-100, delay-200, etc., to apply delays of 100ms, 200ms, etc.

Example:

```
<button class="transition delay-150 duration-300 ease-in-out hover:bg-blue-500">
Hover me
</button>
```

Use the delay-* utilities to control an element’s transition-delay.

2.1.5: Utilizing animation and transitions

Step 1: Browse to the folder where your project is created

Open File Explorer or your terminal and navigate to the folder where your React project is stored.

Step 2: Open command prompt

- Windows: Press Windows + R, type cmd, and press Enter.
- Mac/Linux: Open the Terminal from your applications or use Ctrl + Alt + T.

Step 3: Type the command to open the project in your text editor

Assuming you're using VS Code, type the following command in the terminal:code . This will open the current project folder in VS Code.

Step 3: Access the React component

In VS Code, browse the src folder and open any component file (like App.js or any other component file you wish to edit).

Step 4: Add animation

To add an animation, you can use **Tailwind CSS** classes (assuming Tailwind CSS is already installed in your project).

For example, let's add a bounce animation:

```
<div className="ml-5 animate-bounce"> I bounce! </div>
```

You can also customize animations like this:

```
<div className="animate-bounce duration-300 ease-in-out">
Download
</div>
```

Step 5: Add transitions

Tailwind provides utilities for transitions. Here's an example of how to apply transitions to an element:

```
<button className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-
4 rounded transition duration-300 ease-in-out">
Hover me
</button>
```

This button will smoothly transition to a darker blue when hovered.

2.1.6: Description of flexbox and grid

Tailwind CSS provides a powerful set of utility classes for implementing Flexbox and CSS Grid layouts, allowing for responsive and efficient design without writing custom CSS. Here's a breakdown of how both Flexbox and Grid can be utilized using Tailwind's utility classes:

Flexbox

Flexbox is designed for one-dimensional layouts, where items are arranged in a row or column. Here are some key Tailwind utility classes for Flexbox:

Container setup:

flex: Makes the element a flex container, enabling the flex behavior for its children.
inline-flex: Applies flex layout to the element but treats it as an inline element.

Example:

```
<div className="flex">...</div>
```

Direction

Controls the direction in which the flex items are laid out:

flex-row: Lays out items in a horizontal row (default).
flex-row-reverse: Reverses the order of items in a row.
flex-col: Lays out items in a vertical column.
flex-col-reverse: Reverses the order of items in a column.

Example: <div className="flex flex-col">...</div>

Alignment:

Controls how flex items are aligned within the container:

For Main Axis (Horizontal) Alignment:

- justify-start: Aligns items to the start of the container.
- justify-center: Centers items horizontally.
- justify-end: Aligns items to the end of the container.

- justify-between: Distributes items evenly with space between them.
- justify-around: Distributes items with equal space around them.
- justify-evenly: Distributes items with equal space between and around them.

Example: <div className="flex justify-center">...</div>

For Cross Axis (Vertical) Alignment:

items-start: Aligns items to the top (start) of the container.

items-center: Centers items vertically.

items-end: Aligns items to the bottom (end) of the container.

items-stretch: Stretches items to fill the container height.

items-baseline: Aligns items based on their text baselines.

Example: <div className="flex items-center">...</div>

✓ Flex wrapping

Controls how flex items wrap within the container:

- **flex-wrap**: Allows items to wrap onto multiple lines.
- **flex-wrap-reverse**: Wraps items in reverse order.
- **flex-nowrap:** Prevents items from wrapping (default).

Example: **<div className="flex flex-wrap">...</div>**

Flex Item Properties

Controls how individual flex items grow, shrink, or maintain their sizes:

- flex-grow: Makes the flex item grow to fill available space.
- flex-shrink: Shrinks the item if necessary.
- flex-none: Prevents the item from growing or shrinking.
- flex-auto: Allows the item to grow and shrink as needed.

Example:

```
<div className="flex">
<div className="flex-grow">Level 3</div>
<div className="flex-shrink">Level 4</div>
</div>
```

Gap

Adds spacing between flex items:

gap-x-{size}: Adds horizontal spacing between items.
gap-y-{size}: Adds vertical spacing between items.
gap-{size}: Adds both horizontal and vertical spacing.
Example: **<div className="flex gap-4">...</div>**

Example of full code:

```
function FlexExample() {
return (
<div className="flex flex-col items-center justify-center min-h-screen">
<div className="flex gap-4">
<div className="flex-grow p-4 bg-blue-500 text-white">Item 1</div>
<div className="flex-shrink p-4 bg-green-500 text-white">Item 2</div>
<div className="p-4 bg-red-500 text-white">Item 3</div>
</div>
</div>
);
}
export default FlexExample;
```

Example:

```
<div class="flex flex-row gap-4">
<div class="basis-1/4 bg-red-500 rounded">Level 3</div>
<div class="basis-1/4 bg-green-500 rounded">Level 4</div>
<div class="basis-1/2 bg-blue-500 rounded">Level 5</div>
</div>
```

✓ Grid

Grid is more suited for two-dimensional layouts, allowing for both rows and columns. Tailwind provides a set of utility classes for Grid as well:

Basic Grid Structure

Tailwind provides utility classes to define grids and control the number of columns and rows. To use the grid system, simply apply the grid class to a container element.

```
<div className="grid">
{/* Grid Items */}
</div>
```

Gap (Spacing)

To add spacing between grid items, you can use the **gap-*** utility. For example, gap-4 will apply a uniform gap between all items.

```
<div className="grid grid-cols-2 gap-4">
<div>Item 1</div>
<div>Item 2</div>
</div>
```

Defining Columns

You can control the number of columns in the grid using the grid-cols-* class. For example, grid-cols-3 will create three equal-width columns.
For example:

```
<div className="grid grid-cols-3 gap-4">
<div>Item 1</div>
<div>Item 2</div>
<div>Item 3</div>
</div>
```

This creates a grid with 3 columns, and the gap-4 adds spacing between the grid items.

Defining Rows

You can define rows using **grid-rows-*** classes. This sets the number of rows in the grid.

```
<div className="grid grid-rows-2 gap-4">
<div>Item 1</div>
<div>Item 2</div>
<div>Item 3</div>
</div>
```

This creates a grid with 2 rows. If the content exceeds the defined rows, it will flow into new rows automatically.

Span Columns and Rows:

You can make grid items span across multiple columns or rows using col-span-* or row-span-*.

```
<div className="grid grid-cols-3 gap-4">
<div className="col-span-2">Item 1 (Spans 2 Columns)</div>
<div>Item 2</div>
<div>Item 3</div>
</div>
```

Auto Rows and Columns

You can use auto-rows-{size} and auto-cols-{size} to control the size of automatically created rows and columns.

```
<div className="grid grid-flow-row auto-rows-min gap-4">
<div>Item 1</div>
<div>Item 2</div>
</div>
```

This ensures that each row takes the minimum space required by its content.

Alignment

Tailwind provides alignment utilities such as justify-items-*, items-center, and place-items-* to control how grid items are aligned within their grid area.

```
<div>Item 1</div>
<div>Item 2</div>
<div>Item 3</div>
</div>
```

Example:

```
const GridExample = () => {
return (
<div className="grid grid-cols-2 gap-4 sm:grid-cols-3 lg:grid-cols-4">
<div className="p-4 bg-blue-500">Item 1</div>
<div className="p-4 bg-blue-500">Item 2</div>
<div className="p-4 bg-blue-500">Item 3</div>
<div className="p-4 bg-blue-500">Item 4</div>
</div>
);
};
export default GridExample;
```