CSS – Cascade Style Sheet

Il CSS nasce dall'esigenza di separare la struttura semantica di un documento HTML dalla sua rappresentazione visiva, questo garantisce manutenibilità e modularità del codice. Nel modello a cascata, le regole di stile vengono elaborate seguendo una gerarchia precisa che considera origine, specificità e ordine d'inclusione; tale meccanismo assicura che le modifiche locali non compromettano l'intero impianto stilistico, promuovendo un'architettura di design che si adatta con agilità a evoluzioni future.

Il box model

Alla base di ogni layout vi è il cosiddetto box model, che concettualizza ogni elemento come un contenitore composto da:
- un'area di contenuto
- da uno spazio interno definito dal padding
- da una cornice
- da uno spazio esterno denominato margine.

Adottare il box-sizing border-box

```
* {
     box-sizing: border-box;
}
```

significa includere padding e bordo nel calcolo delle dimensioni di un elemento, semplificando i calcoli e garantendo un maggior controllo sul posizionamento.

Ex:

```
.example {
     width: 200px;
     padding: 20px;
     border: 5px solid;
}
```

il contenitore rimane largo esattamente 200 pixel, anziché superare la misura attesa di 250 pixel.

Dal mobile-first alle media query

Con l'avvento dei dispositivi mobili e delle viewport di dimensioni variabili, il CSS ha dovuto evolversi oltre il layout statico basato su float o posizionamenti assoluti. Le media query introducono una logica condizionale che permette di modulare le regole in funzione delle caratteristiche del dispositivo, creando un linguaggio di tipi (feature queries) che si comportano come switch all'interno del foglio di stile.

```
@media (min-width: 768px) {
  .container { display: grid; grid-template-columns: 1fr 2fr; }
}
```

Il potere dichiarativo di Flexbox e Grid

Flexbox e Grid rappresentano due paradigmi diversi ma complementari: il primo si concentra sulla gestione di una singola dimensione organizzando gli elementi lungo un asse principale e uno secondario, il secondo proietta il layout in una griglia bidimensionale in cui le righe e le colonne formano una matrice esplicita.
Teoricamente, Flexbox incapsula un modello di distribuzione spaziale in termini di crescita e contrazione, governato dalle proprietà di flex-grow, flex-shrink e flex-basis, conciliando semplicità d'uso e dinamismo.

Grid, al contrario, introduce l'idea di celle nominate e tracciati espliciti, così da poter definire aree logiche e gestire posizionamenti complessi in maniera dichiarativa, sfruttando unità frazionarie (fr) e funzioni come minmax() per schedulare l'allocazione automatica dello spazio residuo.

Unità di misura: tra costanza e adattabilità

La convivenza di unità fisse e relative è la chiave per un design veramente adattivo: mentre i valori in pixel garantiscono precisione in contesti controllati, unità quali **em**, **rem**, **vw e vh** permettono di creare geometrie fluide che rispondono ai cambiamenti del viewport e alla tipografia.

La scelta tra **em e rem**, ad esempio, incide su come l'eredità delle dimensioni del font si propaga nella gerarchia del ***DOM***, offrendo strategie diverse per modulare spazi e tipografie in modo coerente.

Selettori avanzati e gestione della specificità

Sul fronte della selettività e della specificità, i selettori avanzati ampliano il potere espressivo permettendo di colpire elementi in base ad attributi, relazioni strutturali o posizioni precise nel flusso del documento.

L'impiego di **nth-child e nth-of-type**, pur consentendo layout di grande precisione, introduce una dipendenza dal markup che può rivelarsi fragilizzante in presenza di DOM dinamici. I selettori di attributo offrono un approccio più semantico, ideali per applicare stili in funzione dei dati associati agli elementi, pur con un costo di performance che va ponderato su alberi DOM di grande complessità.

Introduzione alla specificità

La specificità rappresenta il principio fondamentale che consente al motore di CSS di risolvere conflitti quando più regole mirano a uno stesso elemento e proprietà.

Attraverso un sistema di "punteggio" interno, la specificità assegna un valore numerico a ciascun selettore, garantendo che, in caso di collisione, prevalga sempre la regola con il valore maggiore.

Il calcolo della specificità

Ogni selettore viene analizzato secondo quattro categorie cui corrispondono pesi predefiniti: gli stili inline (1 000 punti), gli ID (100), le classi, gli attributi e le pseudo-classi (10), gli elementi e le pseudo-elementi (1). Il motore CSS somma questi valori per ogni componente del selettore, generando un numero finale che definisce la precedenza. Ad esempio, dato il selettore

ex:

ul#menu li.item:hover::before { … }

il calcolo sarà così suddiviso: 1 ID (100) + 1 classe (10) + 1 pseudo-classe (10) + 1 elemento ul + 1 elemento li + 1 pseudo-elemento ::before (tutti insieme valgono 3) = 123. Qualsiasi altra regola con un punteggio inferiore non potrà mai avere la meglio su questa.

Quando definiamo

```
p {
   color: grey;
}
```

Il foglio assegna un punteggio pari a 1, perché abbiamo un solo selettore di tipo elemento.

Se aggiungiamo;

```
p.intro {
     color:black;
}
```

La specificità diventa 11 perché sommando 10 per la classe e 1 per l'elemento. Quindi assume priorità rispetto alla regola precedente.

Se aggiorniamo:

<p id="main" class="intro" style="color: red">

il valore inline (1000) sovrasta ogni altra regola, tingendo il paragrafo di rosso incondizionalmente.

Risoluzione dei conflitti a pari specificità

Quando due o più regole ottengono lo stesso punteggio, entra in gioco la regola dell'ordine di apparizione: vince quella dichiarata per ultima nel foglio di stile o nell'elenco dei fogli inclusi. Di conseguenza, se abbiamo due selettori con specificità 11 definiti in momenti diversi, la regola più "recente" nel sorgente avrà la meglio, anche se il punteggio è identico.

PSEUDO-CLASS SELECTORS

I selettori basati su pseudo-classi ti permettono di "vedere" l'elemento in un dato momento, come se ascoltassi il suo stato, e di reagire con il CSS di conseguenza. Immagina di voler rendere più vivace un link non appena ci passi sopra il mouse: basta usare a:hover per cambiare colore o aggiungere un'ombra. In questo modo, lo stile non è più statico ma risponde all'azione dell'utente, creando un'interazione immediata che rende la pagina molto più coinvolgente.

Selector:pseudo-class {

…

}

Allo stesso modo, quando premi un pulsante e vuoi che il suo aspetto dia l'idea di essere "schiacciato", entra in gioco il selettore :active. Diversamente da :hover, che si attiva al semplice movimento del cursore, :active persiste solo per il tempo in cui il tasto del mouse rimane tenuto premuto, conferendo al bottone un feedback tattile visivo. Se poi devi concentrare l'attenzione su un campo di input quando vi clicchi dentro, il selettore :focus: sottolinea, evidenzia o ingrandisce l'elemento interessato, guidando l'utente nel compilare il form in modo più chiaro.

Parliamo poi delle pseudo-classi dedicate ai moduli. Pensale come una coppia di sensori: da un lato :disabled e :enabled ti segnalano se un controllo è attivo o meno, dall'altro :checked segue lo stato delle checkbox e dei radio button, mentre :valid e :invalid si occupano di far suonare il campanello d'allarme quando l'input non rispetta il pattern desiderato. Così, senza una riga di JavaScript, puoi disegnare in rosso i campi sbagliati o abilitare solo quando tutto è corretto, con un'esperienza utente pulita e reattiva.

Esiste poi un'altra famiglia di pseudo-classi che agisce sulla posizione degli elementi nel loro contenitore. Se vuoi applicare uno stile particolare al terzo elemento di una lista o al primo paragrafo di ogni sezione, :nth-of-type() e :first-of-type ti

danno il potere di scegliere esattamente quale elemento vestire di uno stile unico. In questo modo, puoi creare layout più dinamici e diversificati senza dover aggiungere classi extra al markup HTML.

PSEUDO-ELEMENTS

I pseudo-elementi sono come piccoli "ritagli" di un elemento HTML che puoi prendere, stilizzare e perfino riempire con nuovo contenuto senza toccare il markup originale. Pensali come un modo per intervenire su una parte specifica di un paragrafo, di una lista o persino su quello che l'utente seleziona con il mouse, rendendo il tuo CSS ancora più raffinato e potente. Il loro segreto sta nella sintassi a due punti doppi: anziché scrivere :hover o :active, userai ::first-letter, ::first-line, ::marker o persino ::before e ::after per dire al browser esattamente su quale "pezzo" dell'elemento intervenire.

Immagina di voler cambiare soltanto la prima lettera di ogni punto in un elenco, rendendola più grande e colorata in corallo: applicherai li::first-letter e il browser, indipendentemente da come modifichi il testo o da quanto sia lunga la lista, saprà sempre valorizzare quel singolo carattere iniziale. Se invece desideri sottolineare l'incipit di ogni riga di un paragrafo, li::first-line ti permette di evidenziare la prima riga indipendentemente dall'ampiezza della finestra: così il tuo design resterà coerente anche quando l'utente ridimensiona lo schermo.

Passando agli elenchi puntati, il pseudo-elemento ::marker ti offre la possibilità di prendere il punto elenco stesso e trasformarlo in qualunque simbolo o colore desideri, magari sostituendo il classico pallino con un segno a freccia color fiordaliso, senza aggiungere nemmeno un carattere HTML in più. E non è finita: grazie a ::before e ::after, puoi far comparire testi o simboli prima o dopo qualsiasi elemento idoneo, come quando vuoi anteporre la parola "Consiglio:" a un paragrafo di suggerimenti culinari, o appendere un punto esclamativo alla fine di ogni riga importante, il tutto senza alterare il DOM.

commony used selectors:

https://www.geeksforgeeks.org/blogs/10-css-selectors-every-developer-should-know/

references css https://www.w3schools.com/cssref/index.php

EFFECTS

Le animazioni sul web trasformano semplici pagine in esperienze più coinvolgenti, dando vita a pulsanti che reagiscono al passaggio del cursore, gallerie che scorrono con fluidità e persino effetti di parallasse che suggeriscono profondità. Partendo dalle prime GIF animate e dai plugin proprietari come Flash, oggi possiamo affidarci a CSS nativo per creare transizioni leggere e micro-interazioni senza appesantire il codice o rallentare il caricamento. Sebbene le librerie JavaScript specializzate offrano controlli più raffinati, saper sfruttare al meglio hover, trasformazioni, keyframe e pseudo-classi di CSS garantisce performance ottimali e tempi di sviluppo ridotti. Il vero segreto sta nell'equilibrio: qualche animazione ben calibrata attira l'attenzione e guida l'utente, mentre un uso eccessivo rischia di distrarre o appesantire la pagina. In definitiva, padroneggiare gli effetti in CSS permette di costruire interfacce intuitive, efficaci e piacevoli da esplorare.

l'**hover effect**, che fa cambiare l'aspetto di un elemento al passaggio del cursore

- la modifica del **puntatore del mouse**, per sostituire la freccia standard con cursori animati o colorati
- le **sliding galleries** o **slide show**, utili per scorrere sequenze di immagini o box di testo
- gli **sfondi video** a tutto schermo, che aggiungono movimento immediato alla parte alta della pagina
- l'**effetto parallasse**, che crea l'illusione di profondità muovendo a velocità diverse primo piano e sfondo
- il pulsante di **"back to top"**, che compare mentre scorri e ti riporta in cima alla pagina
- le **transizioni di elementi e colori** attivate allo scroll, per enfatizzare al volo sezioni o componenti
- lo **fullscreen snapping**, che fa "scattare" lo viewport da un'area all'altra mentre l'utente scorre

TEXT EFFECTS

Quando un blocco di testo non riesce a stare entro i limiti del suo "contenitore", senza una regola precisa, il browser allarga il box o sfonda i bordi..

Con la proprietà text-overflow riacquisti il controllo: ti basta specificare overflow: hidden; white-space: nowrap; text-overflow: ellipsis; per far sì che le parole tagliate vengano sostituite da "…"

In pratica il flusso del testo rimane tutto su un’unica riga (grazie a white-space: nowrap), il contenuto in eccesso viene nascosto (overflow: hidden), e infine appare l’ellissi (text-overflow: ellipsis), un chiaro indizio per l’utente che c’è altro da leggere. Se invece preferisci che il testo venga “troncato” senza punti di sospensione, sostituisci ellipsis con clip e l’eccesso sparirà semplicemente, pur rimanendo accessibile — per esempio con un tooltip al passaggio del mouse.

Spesso vogliamo giocare con l’orientamento del testo, specialmente in layout grafici o per menu verticali. Qui entra in gioco writing-mode. Impostando writing-mode: vertical-rl; il flusso del testo ruota di 90°, partendo da destra verso sinistra: il browser tratta ogni lettera come “in piedi” anziché “sdraiata”, generando colonne di caratteri. È utilissimo per titoli da affiancare a immagini o per effetti visivi che spezzano la monotonia di paragrafi troppo ampi.

Quando hai paragrafi pieni di termini molto lunghi — URL, parole composte, codici — e non vuoi abilitare lo scroll orizzontale, la proprietà word-wrap (oggi spesso usata come alias overflow-wrap) ti soccorre. Con word-wrap: break-word; il browser può “spezzare” una parola alla fine della riga se necessario, distribuendo il suo testo su più righe. Così eviti barre di scorrimento orizzontali indesiderate pur senza aggiungere manualmente trattini o spazi supplementari.

Infine, per dare profondità e “vita” ai caratteri, text-shadow è il tuo migliore amico. Si configura con quattro valori: offset orizzontale, offset verticale, raggio di sfocatura e colore. Ad esempio:

```
h1 {
  text-shadow: 2px 2px 4px rgba(0,0,0,0.5);
}
```

TRANSFORM E TRANSITION

Il vero potere delle animazioni CSS sta proprio nell'abbinamento di transform e transition: il primo ti consente di spostare, ruotare, ridimensionare o inclinare un elemento, il secondo controlla la durata e la modalità di quell'animazione, trasformando un cambiamento istantaneo in un movimento fluido.

Per creare animazioni davvero complesse (come un orologio che segna le ore in tempo reale) non basta più una semplice transition: serve la proprietà animation insieme alla regola @keyframes, che ti permette di definire gli "scatti" (key-frames) intermedi della tua animazione. Ecco come funziona in dettaglio.

1. Sintassi di base

@keyframes

Con @keyframes nomeAnimazione { … } definisci i vari step di trasformazione. Puoi usare percentuali (0%, 50%, 75%, 100%) o le keywords from (equivalente a 0%) e to (100%).

```
@keyframes cycle {
  from { transform: rotate(0deg); }
  to   { transform: rotate(360deg); }
}
```

Proprietà animation

La proprietà animation (shorthand) accorpa questi sub-valori:

1. **animation-name**: il nome richiamato da @keyframes
2. **animation-duration**: durata di un ciclo (es. 6s)
3. **animation-timing-function**: il ritmo dell'animazione (es. linear, ease-in-out)
4. **animation-delay**: ritardo prima di partire (es. 1s)
5. **animation-iteration-count**: quante volte ripetere (es. infinite)
6. **animation-direction**: avanti, indietro, alternato (es. alternate)
7. **animation-fill-mode**: cosa succede prima e dopo l'animazione (es. forwards)

Puoi scriverli in una riga:

```
.element {
  animation: cycle 6s linear infinite;
```

```
}
```

2. Esempio completo: l’orologio che gira

Qui mostriamo come costruire un semplice quadrante con lancette che ruotano a velocità diverse.

HTML

```
<div class="clock">
  <div class="hand minutes"></div>
  <div class="hand hours"></div>
</div>
```

CSS

```
/* Contenitore orologio */
.clock {
  position: relative;
  width: 200px;
  height: 200px;
  border: 8px solid #333;
  border-radius: 50%;
  background: #eef;
  margin: 2rem auto;
}

/* Stile comune alle lancette */
.hand {
  position: absolute;
  bottom: 50%;
  left: 50%;
  transform-origin: center bottom;
  background: #c33;
}

/* Lancetta dei minuti */
.minutes {
  width: 4px;
  height: 80px;
  animation: cycle 6s linear infinite;
}

/* Lancetta delle ore (più corta e lenta) */
.hours {
  width: 6px;
  height: 60px;
  animation: cycle 60s linear infinite;
}

/* Definizione dei keyframes */
@keyframes cycle {
  from { transform: rotate(0deg); }
  to   { transform: rotate(360deg); }
}
```

Spiegazione

- Entrambe le lancette usano animation: cycle … infinite;
- I minuti fanno un giro completo in **6 secondi**, le ore in **60 secondi**
- linear garantisce velocità costante
- @keyframes cycle applica transform: rotate() da 0° a 360°

3. Altri esempi di animazioni avanzate

3.1 Effetto "bounce" con più keyframes

```
@keyframes bounce {
  0%, 100% {
    transform: translateY(0);
  }
  50% {
    transform: translateY(-30px);
  }
}

.bouncy {
  display: inline-block;
  padding: 1rem;
  background: lightseagreen;
  color: white;
  animation: bounce 2s ease-in-out infinite;
}
```

Uso in HTML

```
<div class="bouncy">Rimbalzo!</div>
```

3.2 Sequenza di colori

```
@keyframes rainbow {
  0%   { background-color: red; }
  25%  { background-color: yellow; }
  50%  { background-color: lime; }
  75%  { background-color: cyan; }
  100% { background-color: magenta; }
}

.rainbow-box {
  width: 150px;
  height: 150px;
  animation: rainbow 5s ease infinite;
}
```

Uso in HTML

```
<div class="rainbow-box"></div>
```

Fotogrammi chiave: da "from→to" a passi intermedi

- **`from` e `to`** sono le scorciatoie per indicare rispettivamente l'inizio (0%) e la fine (100%) dell'animazione.
- **Percentuali intermedie** (`25%`, `50%`, `75%`, ecc.) ti consentono di inserire punti di svolta in cui modificare proprietà diverse: puoi, ad esempio, far crescere la larghezza fino a metà animazione, poi cambiare il colore e infine riportare tutto allo stato iniziale.

Questo meccanismo è ciò che rende `@keyframes` così flessibile: a ogni step puoi intervenire su una o più proprietà, creando movimenti articolati e transizioni multiple in una sola dichiarazione.

Collegamento tra `@keyframes` e `animation`

Dopo aver definito la sequenza con:

```
@keyframes nomeAnimazione {
  0%   { /* stato iniziale */ }
  50%  { /* step intermedio */ }
  100% { /* stato finale */ }
}
```

ti basta assegnarla a un elemento con la proprietà `animation-name: nomeAnimazione;` (o usando la sintassi shorthand `animation: nomeAnimazione durata timing infinite;`). In questo modo il browser sa:

1. **Quale sequenza** (il nome)
2. **Quanto duri** l'animazione
3. **Con quale ritmo** (timing function)
4. **Quante volte** ripeterla (iteration count)
5. **Se e come** mantenere lo stato finale (`animation-fill-mode`)

Vantaggi di `@keyframes`

- **Controllo preciso**: non sei vincolato a un solo cambiamento, ma puoi suddividere l'animazione in tanti step logici.
- **Proprietà multiple**: in un singolo keyframe puoi animare larghezza, posizione, colore, rotazione, opacità e via dicendo, anche in combinazione.
- **Loop, alternanze e fill modes**: con sub-proprietà come `animation-iteration-count`, `animation-direction: alternate` e `animation-fill-mode` puoi far ripetere l'animazione all'infinito, farla ritornare indietro o mantenere l'aspetto dell'ultimo frame.

ANIMATION AND EFFECTS CHEAT SHEET

Trasformazioni spaziali con `transform`

La proprietà `transform` ti consente di **sospendere** lo stile statico di un elemento e di **agire** su di esso nel suo spazio 2D o 3D. Puoi pensare a `transform` come a un piccolo sotto-linguaggio dentro il CSS che ti permette di:

- **Ruotare** (`rotate`, `rotateX/Y/Z`, `rotate3d`)
- **Spostare** (`translate`, `translateX/Y/Z`, `translate3d`)
- **Scalare** (`scale`, `scaleX/Y/Z`, `scale3d`)
- **Inclinare** (`skew`, `skewX`, `skewY`)

```
.sample {
  /* ruota di 45°, ingrandisci del 50% e trasla di 45px a destra */
  transform: rotate(45deg) scale(1.5) translate(45px);
  /* punto di ancoraggio della trasformazione */
  transform-origin: right bottom;
}
```

Micro-animazioni con `transition`

Quando vuoi che il passaggio da uno stato di stile a un altro non sia istantaneo, ma graduale, usi `transition`. Questa proprietà shorthand raggruppa:

1. **transition-property** (quale proprietà animare)
2. **transition-duration** (in quanto tempo)
3. **transition-timing-function** (curva di velocità: `ease`, `linear`, `ease-in-out`…)
4. **transition-delay** (ritardo prima di partire)

```
.box {
  transition: transform 0.5s ease-in-out 0.2s;
}
.box:hover {
  transform: translateX(50px);
}
```

Animazioni complesse con `animation` e `@keyframes`

Quando devi orchestrare **più fasi**, cambiare **diverse proprietà** in momenti prestabiliti, o ripetere un ciclo all'infinito, passi a `animation`:

- **`@keyframes nomeAnimazione { … }`** definisce i tuoi fotogrammi-chiave:
  - `from`/`to` (0% → 100%)
  - Oppure serie di percentuali intermedie (30%, 50%, 75%)
  - In ciascun blocco specifichi quali proprietà cambiano e come
- **`animation`** (shorthand) lega l'elemento alla sequenza di keyframes e ne regola:
  - **nome** (`animation-name`)
  - **durata** (`animation-duration`)
  - **funzione di timing** (`animation-timing-function`)
  - **delay** (`animation-delay`)
  - **ripetizioni** (`animation-iteration-count`, es. `infinite`)
  - **direzione** (`animation-direction`, es. `alternate`)
  - **fill-mode** (`animation-fill-mode`, es. `forwards`)
  - **play-state** (`animation-play-state`, es. `paused`/`running`)

```
@keyframes pulse {
  0%, 100% { transform: scale(1); }
  50%      { transform: scale(1.2); }
}

.button {
  animation: pulse 2s ease-in-out infinite;
}
```

`Perché usare un preprocessore?`

1. **Variabili:** invece di ripetere lo stesso codice, assegni valore una sola volta e lo riutilizzi.
2. **Nesting dei selettori:** annidi i tuoi stili proprio come strutturi l'HTML, riducendo la ripetizione di nomi e rendendo il foglio più leggibile.
3. **Mixins e funzioni:** crei blocchi di regole riutilizzabili (con `@mixin` e `@include` in Sass/SCSS) o funzioni (in Stylus) che calcolano valori al volo, per esempio graduazioni di colore o somme di misure.
4. **Controlli di flusso:** cicli (`for`, `each`) e condizionali (`if`) ti aiutano a generare serie di stili o ad applicare regole solo quando serve, senza scrivere tutto a mano.
5. **Import e modularità:** separi il tuo CSS in tanti piccoli file (variabili, tipografia, layout, animazioni…) e poi li "compili" in un unico `main.css`, evitando problemi di ordine di caricamento.

CSS Browser Compatibility Cheat Sheet

Cos'è la compatibilità cross-browser?

È la capacità di un sito web di apparire e comportarsi allo stesso modo su diversi browser (Chrome, Firefox, Safari, Edge, Opera…).

Tipo di problema	Descrizione
Differenze nei motori di rendering	Es. WebKit (Safari), Blink (Chrome/Edge), Gecko (Firefox)
Default CSS diversi	Font, margini, padding, allineamenti variano tra browser
Nuove proprietà non ancora supportate	I browser adottano le nuove specifiche a velocità diverse
Differenze su Flex/Grid	Layout animati non fluidi, problemi di allineamento, proporzioni errate

Strumento	Funzionalità
Can I use	Verifica supporto proprietà CSS nei vari browser/versioni
BrowserStack	Ambiente virtuale per test su browser e dispositivi reali
CrossBrowserTesting	Simile a BrowserStack, include test automatizzati
CSS Validation Service	Controlla la validità del tuo CSS

Media Queries per browser specifici

```
/* Solo per WebKit (Chrome, Safari, Opera) */


@media screen and (-webkit-min-device-pixel-ratio: 0) {
  p {
    color: red;
  }
}
```

`device-pixel-ratio` è una proprietà che indica il rapporto tra **pixel fisici** e **pixel CSS**.

- `-webkit-min-device-pixel-ratio` è la variante **non standard** usata nei browser WebKit.
- Il valore `0` è il più basso possibile, quindi **qualsiasi valore >0** (ovvero qualunque browser WebKit) passerà questa condizione.

```
/* Solo per Safari (via hack non ufficiale) */
@media not all and (min-resolution:.001dpcm) {
  @supports (-webkit-appearance:none) {
    p {
      background: yellow;
    }
  }
}
```

librerie utili:

Reset.css

modernizer

Normalize.css