LESS

(/https://www.html.it/articoli/less-css-programmare-gli-stili/)

analizziamo la sintassi che compete il compilatore disponibile da **lesscss.org(sito di riferimento per la documentazione):** **less.js**.

Per cominciare, vediamo come utilizzare LESS nelle nostre pagine. È molto semplice, dobbiamo importare il codice del compilatore e un foglio di stile di base:

<link rel="stylesheet/less" type="text/css" href="styles.less">

<script src="less.js" type="text/javascript"></script>

Come si vede l'attributo `rel="stylesheet"` è esteso avendo scritto `rel="stylesheet/less"` e subito dopo si aggiunge un js che compila a runtime il file .less.

Ovviamente questa pratica è ottima quando state sviluppando il sito web, ma è più che deprecabile se il sito è in produzione! In quel caso dovrete passare per un'estensione della vostra procedura di **deploy**, che unisca anche la compilazione e possibile minificazione del file css di output.

Per Windows ci sono diverse soluzioni come **WinLess** e **lessc**, ma esiste anche **SimpLess** una soluzione cloud molto comoda.

Una delle cose che salta più velocemente agli occhi, analizzando un file less rispetto al suo omologo compilato CSS è la compattezza.

La notazione di LESSCSS è incredbilmente più compatta:

```
.container {
  …
  .inner {
    …
    h2 {
      …
    }
  }
}
```

```
.container {
  …
  .inner {
    …
    h2 {
      …
      // h2 end
    }
    // inner end
  }
  // container end
}
```

possiamo **inserire i commenti per una singola riga** con il doppio slash (`//`), come in Javascript.

Le variabili

Se abbiamo la sana abitudine di definire un commento iniziale in cui "dichiarare" le color palette del documento, le font-family, e le altre proprietà visive, ci sarà subito familare l'uso che consigliamo: all'inizio del vostro file .less è molto sensato, ma non obbligatorio, **definire tutte le variabili**, con la seguente sintassi:

```
@nomevariabile = VALORE
```

Operatori

```
+ - * /
```

Esempio:

```
@base-font-size: 13px;
```

```
.selector {
  font-size: @base-font-size + 5px;
}
```

I mixin

Un altro vantaggio veramente interessante di LESSCSS è quello di poter **definire dei mixin**, parametrici e non. In sostanza si tratta di avere un blocco che può essere ripreso in maniera molto compatta durante lo sviluppo:

```
.fancy-font {
  font-family: "CustomFamily", Arial, sans-serif;
  font-size:16px;
  color: @primary-color;
}
```

Possiamo certamente assegnarla a un elemento HTML nel classico modo `<h2 class="fancy-font">` ma la cosa più interessante è un'altra.

```
#header {
    h1,h2,h3 {
      .fancy-font;
    }
    h1 {
        font-size: @base-font-size + 13px;
    }
    h2 {
        font-size: @base-font-size + 7px;
    }
}
```

Mixin e funzioni

La naturale e più potente estensione dei mixin si verifica quando li utilizziamo in combinazione con le variabili, **trasformando di fatto i mixin in vere funzioni.**

```
.fancy-font(@header-size: 20px, @color, @padding) {
  font-family: "CustomFamily", Arial, sans-serif;
  font-size:@header-size;
  color: @color;
  padding: @padding;
}
```

```
.fancy-font(30px, #ff0000, 10px);
.fancy-font(20px, #00ff00, 10px);
.fancy-font(10px, @secondary-color, 10px);
```

Risultato compilato:

```css
#header h1 {
  font-family: "CustomFamily", Arial, sans-serif;
  font-size: 30px;
  color: #ff0000;
  padding: 10px;
}
#header h2 {
  font-family: "CustomFamily", Arial, sans-serif;
  font-size: 20px;
  color: #00ff00;
  padding: 10px;
}
#header h3 {
  font-family: "CustomFamily", Arial, sans-serif;
  font-size: 10px;
  color: #ff00ff;
  padding: 10px;
}
#header h1 {
  font-size: 26px;
}
#header h2 {
  font-size: 20px;
}
```

È chiaro che si possono anche mischiare le due tecniche di mixin: se infatti abbiamo l’abitudine di definire:

```css
.heading {
  margin:0;
  font-weight:700;
  text-transform:uppercase;
}
```

```
.fancy-font(@header-size: 20px, @color, @padding) {
  font-family: "CustomFamily", Arial, sans-serif;
  font-size:@header-size;
  color: @color;
  padding: @padding;
  .heading;
}
```

```
.heading {
  margin: 0;
  font-weight: 700;
  text-transform: uppercase;
}
#header h1 {
  font-family: "CustomFamily", Arial, sans-serif;
  font-size: 30px;
  color: #ff0000;
  padding: 10px;
  margin: 0;
  font-weight: 700;
  text-transform: uppercase;
}
#header h2 {
  font-family: "CustomFamily", Arial, sans-serif;
  font-size: 20px;
  color: #00ff00;
  padding: 10px;
  margin: 0;
  font-weight: 700;
  text-transform: uppercase;
}
#header h3 {
  font-family: "CustomFamily", Arial, sans-serif;
  font-size: 10px;
  color: #ff00ff;
  padding: 10px;
  margin: 0;
```

Altre interessanti fattispecie da analizzare per LESSCSS è quella delle **funzioni di modifica di colore:**

```
lighten(@color, 10%);     // return a color which is 10% *lighter* than @color
darken(@color, 10%);      // return a color which is 10% *darker* than @color
saturate(@color, 10%);    // return a color 10% *more* saturated than @color
desaturate(@color, 10%);  // return a color 10% *less* saturated than @color
fadein(@color, 10%);      // return a color 10% *less* transparent than @color
fadeout(@color, 10%);     // return a color 10% *more* transparent than @color
spin(@color, 10);         // return a color with a 10 degree larger in hue than @colo
spin(@color, -10);        // return a color with a 10 degree smaller hue than @color
```

Namespace e scope delle variabili

Per quanto riguarda il primo concetto, è chiaro che un namespace ha un carattere di organizzazione del codice

```
#bundle {
  .button () {
    display: block;
    border: 1px solid black;
    background-color: grey;
    &:hover { background-color: white }
  }
  .tab { ... }
  .citation { ... }
}
```

Se volessimo assegnare a un `#header` a solo le proprietà di `button()` dentro `#bundle` potremmo scrivere

```
#header a {
  color: orange;
  #bundle > .button;
}
```

Questo fa in modo di "pescare" solo le proprietà che ci interessano e null'altro.

Il concetto di **scope** è invece più legato alla "validità" delle variabili nei blocchi di codice, un po' come succede nei linguaggi di programmazione tradizionali. Vediamo un esempio:

```
@var: red;
#page {
  @var: white;
  #header {
    color: @var; // white
  }
}
#footer {
  color: @var; // red
}
```

Import

È spesso necessario importare dei css all'interno del proprio foglio di stile. Ebbene con questa strategia quel che vi consigliamo di fare è avere un file separato che sia di definition di variabili che poi importerete con una notazione identica a quella di CSS.

```
@import "definition.less";
```

Stringhe

utile definire come variabili anche delle stringhe

supponiamo di dover spostare un progetto cambiando il file system dal server di sviluppo al server di produzione o per qualunque altra ragione che rende non perfettamente allineati gli ambienti, in questi casi definire:

```
@base-url: "http://www.example.com";
background-image: url("@{base-url}/images/bg.png");
```

Less using npm

Usiamo visual studio code:

utilizzeremo il package npm less per compilare i file less, lo scarichiamo:

npm istall -g less

(**Qui dove vengono installati sul mio computer gli npm globali**

C:\Users\Luca\AppData\Roaming\npm\node_modules)

poi scarichiamo anche il watcher per lanciare automaticamente la compilazione ogni volta che si salva il file less.

npm install -g less-watch-compiler

se creiamo un file .less e poi lanciamo questa istruzione:

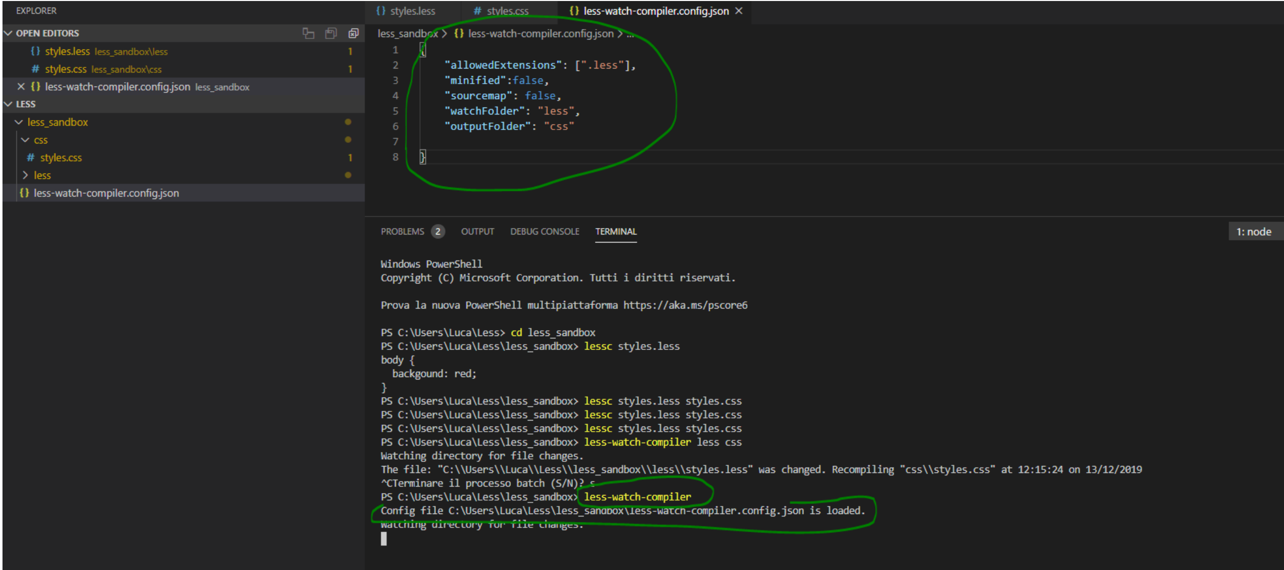
```
PS C:\Users\Luca\Less\less_sandbox> lessc styles.less styles.css
```

Verrà compilato dentro un file css del css regolare sulla base del file less, ogni volta che apportiamo una modifica però per ricompilare nel css gli aggiornamenti fatti nel less dovremo rilanciare il comando.

Con il watcher possiamo automatizzare la compilazione ad ogni salvataggio del file less con questo comando:

```
PS C:\Users\Luca\Less\less_sandbox> less-watch-compiler less css
Watching directory for file changes.
The file: "C:\\Users\\Luca\\Less\\less_sandbox\\less\\styles.less" was changed. Recompiling "css\\styles.css" at 12:15:24 on 13/12/2019
```

se poi creiamo un file di configurazione per il watcher ogni volta che lo lanciamo non dobbiamo specificare nulla perché lui leggerà il file di configurazione.



GULP

Gulp è un task runner per javascript application.

Viene installato con npm (ovviamente node js installato è un prerequisito)

Gulp evita compiti ripetitivi automatizzandoli e si serve di centinaia di plugin per lanciare questi task.

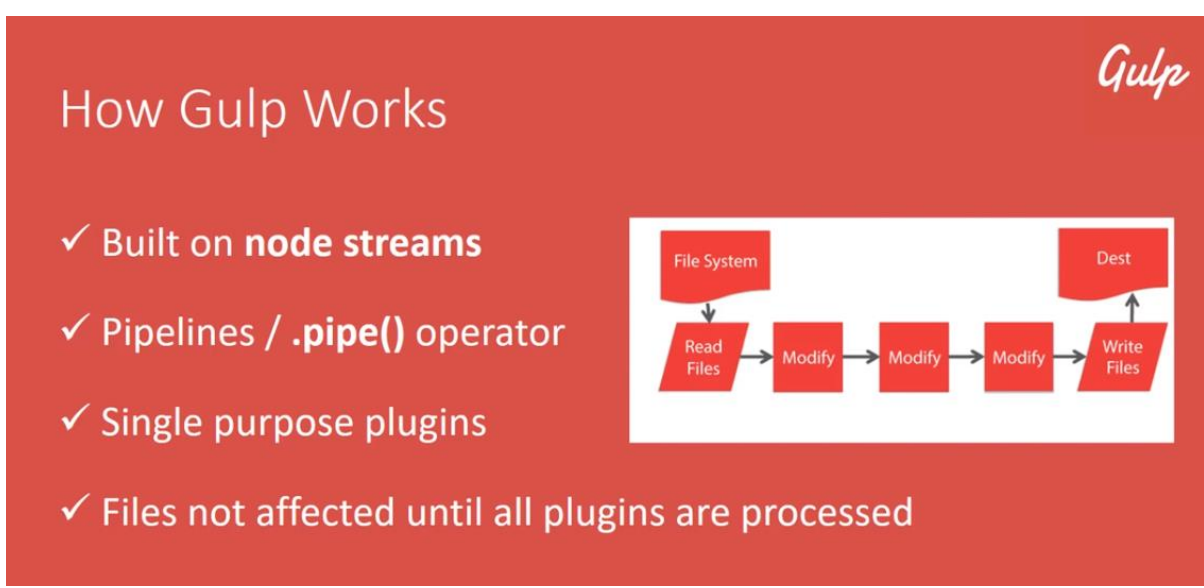
Fra i task più comuni:

minificazione dei css e js files

concatenazione (di più file in un unico file)

ottimizzazioni eccetra

vediamo come lavora gulp:



Gulp lavora sopra uno nodo che monitora costantemente, il node stream è un continuo flusso di dati che vengono manipolati attraverso la pipe lines che è una catena di processi percui l'output prodotto da uno è l'input del successivo processo.

Installiamo quindi gulp:

npm install -g gulp

lanciamo un comando per creare un file package.json di partenza:

npm init

lanciamolo dal terminale di visual studio code:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS D:\L_SCHIAVON_DOCUMENTS\LAVORO\VISUALSTUDIOCODE\gulpexapp> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (gulpexapp)
version: (1.0.0)
description: example app using gulp
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to D:\L_SCHIAVON_DOCUMENTS\LAVORO\VISUALSTUDIOCODE\gulpexapp\package.json:

{
  "name": "gulpexapp",
  "version": "1.0.0",
```

questo comando crea di default:

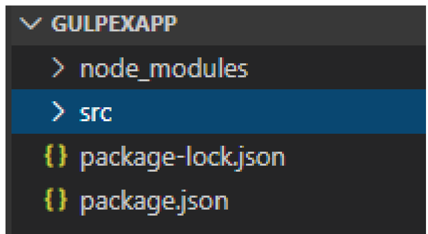
```
EXPLORER
OPEN EDITORS
 × {} package.json
GULPEXAPP
 {} package.json

{} package.json ×
{} package.json > ...
1  {
2    "name": "gulpexapp",
3    "version": "1.0.0",
4    "description": "example app using gulp",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "author": "",
10   "license": "ISC"
11 }
12
```

Ora installiamo gulp localmente (lo abbiamo installato solo globalmente)

Ed in più installiamo le dipendenze da sviluppatore:

```
npm install --save-dev gulp
```

Questo crea una cartella node modules dentro il percorso in cui viene lanciato, a fianco a questa cartella creiamo una cartella src dove metteremo tutto il sorgente che verrà gestito da gulp e poi creeremo una cartella build o public dove verranno salvati tutti i file compilati.:



Poi creiamo un file gulp.js

Utilizzo dei compilatori sass e less in ambiente visual studio

Visual studio offre una estensione installabile che si chiama **web compiler** la quale compila automaticamente al salvataggio less sass ecc… basta installarla come estensione e creare un file compilerconfig.json in root dentro cui posizionare l'input ed output file ossia la posizione del file less e dove salvare il file css compilato