



Linux
Professional
Institute

Web Development Essentials

Versione 1.0
Italiano

030

Table of Contents

ARGOMENTO 031: SVILUPPO SOFTWARE E TECNOLOGIE WEB	1
 031.1 Fondamenti di Sviluppo Software	2
031.1 Lezione 1	3
Introduzione	3
Codice Sorgente	3
Linguaggi di Programmazione	5
Esercizi Guidati	12
Esercizi Esplorativi	13
Sommario	14
Risposte agli Esercizi Guidati	15
Risposte agli Esercizi Esplorativi	16
 031.2 Architettura di un'Applicazione Web	17
 031.2 Lezione 1	19
Introduzione	19
Client e Server	19
La Parte Client	20
Tipi di Client Web	21
Linguaggi di un Client Web	22
Il Lato Server	24
Gestire i Percorsi dalle Richieste	24
Sistemi di Gestione di Database	25
Gestione del Contenuto	26
Esercizi Guidati	27
Esercizi Esplorativi	28
Sommario	29
Risposte agli Esercizi Guidati	30
Risposte agli Esercizi Esplorativi	31
 031.3 Nozioni Base sull'HTTP	32
031.3 Lezione 1	34
Introduzione	34
La Richiesta del Client	35
L'Intestazione di Risposta	38
Contenuto Statico e Dinamico	40
Caching	41
Le Sessioni HTTP	42
Esercizi Guidati	44
Esercizi Esplorativi	45
Sommario	46

Risposte agli Esercizi Guidati	47
Risposte agli Esercizi Esplorativi	48
ARGOMENTO 032: MARCATURA DI UN DOCUMENTO HTML	49
032.1 Anatomia di un Documento HTML	50
032.1 Lezione 1	51
Introduzione	51
Anatomia di un Documento HTML	51
Intestazione del Documento	55
Esercizi Guidati	59
Esercizi Esplorativi	60
Sommario	61
Risposte agli Esercizi Guidati	62
Risposte agli Esercizi Esplorativi	63
032.2 Semantica HTML e Gerarchia del Documento	66
032.2 Lezione 1	68
Introduzione	68
Il Testo	69
Le Intestazioni	69
Interruzioni di Linea	71
Linee Orizzontali	72
Le Liste in HTML	73
Formattazione del Testo <i>Inline</i>	79
Testo Preformatto	84
Raggruppare gli Elementi	85
Struttura della Pagina HTML	87
Esercizi Guidati	96
Esercizi Esplorativi	97
Sommario	98
Risposte agli Esercizi Guidati	100
Risposte agli Esercizi Esplorativi	102
032.3 Riferimenti HTML e Risorse Integrate	109
032.3 Lezione 1	110
Introduzione	110
Contenuto Integrato	110
Collegamenti	114
Esercizi Guidati	117
Esercizi Esplorativi	118
Sommario	119
Risposte agli Esercizi Guidati	120
Risposte agli Esercizi Esplorativi	121

032.4 Moduli HTML	122
032.4 Lezione 1	123
Introduzione	123
Semplici Moduli HTML	123
Inserimento di grandi quantità di testo: textarea	132
Liste di Opzioni	133
L'Elemento hidden	137
Il Tipo di Input File	137
Bottoni di Azione	138
Azione e Metodi nei Moduli	139
Esercizi Guidati	141
Esercizi Esplorativi	142
Sommario	143
Risposte agli Esercizi Guidati	144
Risposte agli Esercizi Esplorativi	145
ARGOMENTO 033: STILE DEI CONTENUTI CON I CSS	147
033.1 Concetti Base dei CSS	148
033.1 Lezione 1	149
Introduzione	149
Applicare gli Stili	150
Esercizi Guidati	157
Esercizi Esplorativi	158
Sommario	159
Risposte agli Esercizi Guidati	160
Risposte agli Esercizi Esplorativi	161
033.2 Selettori CSS e Applicazione di Stili	162
033.2 Lezione 1	163
Introduzione	163
Stili per l'Intera Pagina	163
Selettori Restrittivi	165
Selettori Speciali	171
Esercizi Guidati	173
Esercizi Esplorativi	174
Sommario	175
Risposte agli Esercizi Guidati	176
Risposte agli Esercizi Esplorativi	177
033.3 Stili nei CSS	178
033.3 Lezione 1	179
Introduzione	179
Proprietà e Valori Comuni dei CSS	179

Colori	179
Sfondo	182
Bordi	184
Valori di Unità	184
Unità Relative	185
Caratteri e Proprietà del Testo	186
Esercizi Guidati	189
Esercizi Esplorativi	190
Sommario	191
Risposte agli Esercizi Guidati	192
Risposte agli Esercizi Esplorativi	193
033.4 Modellazione e Disposizione dei Contenitori nei CSS	194
033.4 Lezione 1	195
Introduzione	195
Flusso Normale	195
Personalizzazione del Flusso Normale	203
Design Responsivo	208
Esercizi Guidati	209
Esercizi Esplorativi	210
Sommario	211
Risposte agli Esercizi Guidati	212
Risposte agli Esercizi Esplorativi	213
ARGOMENTO 034: PROGRAMMAZIONE JAVASCRIPT	214
034.1 Esecuzione e Sintassi in JavaScript	215
034.1 Lezione 1	216
Introduzione	216
Esecuzione di JavaScript nel Browser	216
La Console del Browser	219
Istruzioni in JavaScript	220
I Commenti in JavaScript	221
Esercizi Guidati	224
Esercizi Esplorativi	225
Sommario	226
Risposte agli Esercizi Guidati	227
Risposte agli Esercizi Esplorativi	228
034.2 Strutture di Dati in JavaScript	229
034.2 Lezione 1	230
Introduzione	230
Linguaggi di Alto Livello	230
Dichiarazione di Costanti e Variabili	231

Tipi di Valore	234
Operatori	238
Esercizi Guidati	241
Esercizi Esplorativi	242
Sommario	243
Risposte agli Esercizi Guidati	244
Risposte agli Esercizi Esplorativi	245
034.3 Strutture di Controllo e Funzioni in JavaScript	246
034.3 Lezione 1	247
Introduzione	247
Dichiarazioni If	247
Strutture di Commutazione	252
Cicli	255
Esercizi Guidati	260
Esercizi Esplorativi	261
Sommario	262
Risposte agli Esercizi Guidati	263
Risposte agli Esercizi Esplorativi	264
034.3 Lezione 2	266
Introduzione	266
Definire una Funzione	266
Ricorsione di Funzione	271
Esercizi Guidati	275
Esercizi Esplorativi	276
Sommario	277
Risposte agli Esercizi Guidati	278
Risposte agli Esercizi Esplorativi	279
034.4 Manipolazione con JavaScript del Contenuto e dello Stile di un Sito Web	280
034.4 Lezione 1	281
Introduzione	281
Interagire con il DOM	281
Contenuto HTML	282
Selezionare Elementi Specifici	284
Lavorare con gli Attributi	285
Lavorare con le Classi	290
Gestori di eventi	291
Esercizi Guidati	294
Esercizi Esplorativi	295
Sommario	296
Risposte agli Esercizi Guidati	297

Risposte agli Esercizi Esplorativi	298
ARGOMENTO 035: PROGRAMMAZIONE SERVER CON NODE.JS	299
035.1 Fondamenti di Node.js	300
035.1 Lezione 1	301
Introduzione	301
Per Iniziare	302
Esercizi Guidati	309
Esercizi Esplorativi	310
Sommario	311
Risposte agli Esercizi Guidati	312
Risposte agli Esercizi Esplorativi	313
035.2 Fondamenti di Node.js Express	314
035.2 Lezione 1	316
Introduzione	316
Script di Inizializzazione Server	316
Rotte	319
Adeguamenti alla Risposta	323
Sicurezza dei Cookie	326
Esercizi Guidati	327
Esercizi Esplorativi	328
Sommario	329
Risposte agli Esercizi Guidati	330
Risposte agli Esercizi Esplorativi	331
035.2 Lezione 2	332
Introduzione	332
File Statici	333
Output Formattato	333
I Template	338
Template HTML	340
Esercizi Guidati	344
Esercizi Esplorativi	345
Sommario	346
Risposte agli Esercizi Guidati	347
Risposte agli Esercizi Esplorativi	348
035.3 Fondamenti di SQL	349
035.3 Lezione 1	350
Introduzione	350
SQL	350
SQLite	351
Accedere al Database	352

Struttura di una Tabella	353
Inserimento dei Dati	354
Query	355
Modificare i Contenuti di un Database	356
Chiudere un Database	358
Esercizi Guidati	359
Esercizi Esplorativi	360
Sommario	361
Risposte agli Esercizi Guidati	362
Risposte agli Esercizi Esplorativi	363
Imprint	364



Argomento 031: Sviluppo Software e Tecnologie Web



031.1 Fondamenti di Sviluppo Software

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 031.1

Peso

1

Arese di Conoscenza Chiave

- Comprendere cos’è il codice sorgente
- Comprendere i principi dei compilatori e degli interpreti
- Comprendere il concetto di librerie
- Comprendere i concetti di programmazione funzionale, procedurale e orientata agli oggetti
- Conoscenza delle caratteristiche comuni degli editor di codice sorgente e degli ambienti di sviluppo integrati (IDE)
- Conoscenza dei sistemi di controllo della versione
- Conoscenza dei test del software
- Conoscenza dei principali linguaggi di programmazione (C, C++, C#, Java, JavaScript, Python, PHP)



031.1 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	031 Sviluppo Software e Tecnologie Web
Obiettivo:	031.1 Fondamenti di Sviluppo Software
Lezione:	1 di 1

Introduzione

I primissimi computer erano programmati attraverso un estenuante processo di inserimento di cavi. Gli scienziati informatici iniziarono presto la ricerca di modi più semplici per dare istruzioni ai computer. Questo capitolo introduce agli strumenti di programmazione. Discute i modi chiave attraverso il quale le istruzioni di testo – i linguaggi di programmazione – rappresentano i compiti che un programmatore vuole realizzare, e gli strumenti che trasformano il programma in una forma chiamata *linguaggio macchina* che un computer può eseguire.

NOTE

In questo testo, i termini *programma* e *applicazione* sono usati in modo intercambiabile.

Codice Sorgente

Un programmatore normalmente sviluppa un'applicazione scrivendo una descrizione testuale, chiamata *codice sorgente*, del compito da assegnare alla macchina. Il codice sorgente è in un *linguaggio di programmazione* accuratamente definito che rappresenta ciò che il computer può fare in un'astrazione di alto livello che gli umani possono comprendere. Sono stati sviluppati anche strumenti per permettere ai programmatore e ai non programmatore di esprimere visivamente queste

istruzioni, ma scrivere codice sorgente è ancora il modo predominante di programmare.

Nello stesso modo in cui un linguaggio naturale ha nomi, verbi e costruzioni per esprimere idee in modo strutturato, le parole e la punteggiatura in un linguaggio di programmazione sono rappresentazioni simboliche di operazioni che saranno eseguite dalla macchina.

In questo senso, il codice sorgente non è molto diverso da qualsiasi altro testo in cui l'autore impiega le regole stabilite di un linguaggio naturale per comunicare con il lettore. Nel caso del codice sorgente, il “lettore” è la macchina, quindi il testo non può contenere ambiguità o incoerenze, per quanto sottili.

E come ogni testo che discute in profondità qualche argomento, anche il codice sorgente deve essere ben strutturato e organizzato logicamente quando si sviluppano applicazioni complesse. Programmi molto semplici ed esempi didattici possono essere memorizzati nelle poche righe di un singolo file di testo, che contiene tutto il codice sorgente del programma. Programmi più complessi possono essere suddivisi in migliaia di file, ognuno con migliaia di linee.

Il codice sorgente delle applicazioni professionali dovrebbe essere organizzato in diverse cartelle, di solito associate a uno scopo particolare. Un programma di chat, per esempio, può essere organizzato in due cartelle: una che contiene i file di codice che gestiscono la trasmissione e la ricezione dei messaggi in rete, e un'altra cartella che contiene i file che costruiscono l'interfaccia e reagiscono alle azioni dell'utente. In effetti, è comune avere molte cartelle e sottocartelle con file di codice sorgente dedicati a compiti molto specifici all'interno dell'applicazione.

Inoltre, il codice sorgente non è sempre isolato nei propri file, in un unico linguaggio. Nelle applicazioni web, per esempio, un documento HTML può incorporare codice JavaScript per completare il documento con funzionalità extra.

Editor di codice e IDE

La varietà di modi in cui il codice sorgente può essere scritto può essere impressionante. Pertanto, molti sviluppatori approfittano di strumenti che aiutano a scrivere e testare il programma.

Il file del codice sorgente è solo un semplice file di testo. Come tale, può essere modificato da qualsiasi editor di testo, non importa quanto semplice. Per rendere più facile distinguere tra codice sorgente e testo semplice, ogni linguaggio adotta un'estensione autoesplicativa del nome del file: `.c` per il linguaggio C, `.py` per Python, `.js` per JavaScript, e così via. Gli editor generici spesso comprendono il codice sorgente dei linguaggi popolari abbastanza bene da aggiungere corsivi, colori e rientri per rendere il codice comprensibile.

Non tutti gli sviluppatori scelgono di modificare il codice sorgente in un editor generico. Un *ambiente di sviluppo integrato* (IDE) fornisce un editor di testo insieme a strumenti per aiutare il

programmatore a evitare errori sintattici e incoerenze evidenti. Questi editor risultano particolarmente raccomandati ai programmati meno esperti, ma anche quelli con più esperienza li usano volentieri.

Gli IDE popolari come *Visual Studio*, *Eclipse* e *Xcode* gestiscono in modo intelligente ciò che il programmatore digita, suggerendo spesso le parole da usare (completamento automatico) e verificando il codice in tempo reale. Gli IDE possono anche offrire debugging e test automatici per identificare i problemi ogni volta che il codice sorgente cambia.

Alcuni programmati più esperti optano per editor meno intuitivi come *Vim*, che offrono maggiore flessibilità e non richiedono l'installazione di pacchetti aggiuntivi. Questi programmati usano strumenti esterni, standalone, per aggiungere le caratteristiche che sono integrate quando si usa un IDE.

Gestione del Codice

Sia in un IDE sia usando strumenti standalone, è importante impiegare un qualche tipo di *version control system* (VCS). Il codice sorgente è in costante evoluzione perché i difetti imprevisti devono essere corretti e i miglioramenti devono essere incorporati. Una conseguenza inevitabile di questa evoluzione è che le correzioni e i miglioramenti possono interferire con altre parti di applicazioni in una base di codice ampia. Gli strumenti di controllo di versione come *Git*, *Subversion* e *Mercurial* registrano tutti i cambiamenti fatti al codice e chi ha fatto il cambiamento, permettendo di rintracciare ed eventualmente recuperare da una modifica non riuscita.

Inoltre, gli strumenti di controllo di versione permettono a ogni sviluppatore del team di lavorare su una copia dei file di codice sorgente senza interferire con il lavoro degli altri programmati. Una volta che le nuove versioni del codice sorgente sono pronte e testate, le correzioni o i miglioramenti fatti a una copia possono essere incorporati dagli altri membri del team.

Git, il sistema di controllo di versione più popolare, permette a molte copie indipendenti di un *repository* di essere mantenute da persone diverse, che condividono le loro modifiche come desiderano. Tuttavia, che si usi un sistema di controllo di versione decentralizzato o che sia centralizzato, la maggior parte dei team mantiene un repository di fiducia sul cui codice sorgente e risorse si può fare affidamento. Diversi servizi online offrono la memorizzazione di repository di codice sorgente. I più popolari di questi servizi sono GitHub e GitLab, ma vale la pena menzionare anche Savannah del progetto GNU.

Linguaggi di Programmazione

Esiste una grande varietà di linguaggi di programmazione; ogni decennio ha visto l'invenzione di nuovi. Ogni linguaggio di programmazione ha le sue regole e viene raccomandato per scopi

particolari. Anche se i linguaggi mostrano differenze superficiali nella sintassi e nelle parole chiave, ciò che li distingue veramente sono gli approcci concettuali che li rappresentano, conosciuti come *paradigmi*.

Paradigmi

I paradigmi definiscono le premesse su cui si basa un linguaggio di programmazione, soprattutto per quanto riguarda il modo in cui il codice sorgente dovrebbe essere strutturato.

Lo sviluppatore parte dal paradigma del linguaggio per formulare i compiti che la macchina deve eseguire. Questi compiti, a loro volta, sono espressi simbolicamente con le parole e le costruzioni sintattiche offerte dal linguaggio.

Il linguaggio di programmazione è *procedurale* quando le istruzioni presentate nel codice sorgente sono eseguite in ordine sequenziale, come la sceneggiatura di un film. Se il codice sorgente è segmentato in funzioni o subroutine, una routine principale si occupa di chiamare le funzioni in sequenza.

Il codice seguente è un esempio di linguaggio procedurale. Scritto in C, definisce variabili per rappresentare il lato, l'area e il volume di forme geografiche. Il valore della variabile `side` viene assegnato in `main()`, che è la funzione invocata quando il programma viene eseguito. Le variabili `area` e `volume` sono calcolate nelle subroutine `square()` e `cube()` che precedono la funzione principale:

```
#include <stdio.h>

float side;
float area;
float volume;

void square(){ area = side * side; }

void cube(){ volume = area * side; }

int main(){
    side = 2;
    square();
    cube();
    printf("Volume: %f\n", volume);
    return 0;
}
```

L'ordine delle azioni definite in `main()` determina la sequenza degli stati del programma, caratterizzati dal valore delle variabili `side`, `area` e `volume`. L'esempio termina dopo aver visualizzato il valore di `volume` con l'istruzione `printf`.

D'altra parte, il paradigma della *programmazione orientata agli oggetti* (OOP) ha come caratteristica principale la separazione dello stato del programma in sotto-stati indipendenti. Questi sotto-stati e le operazioni associate sono gli *oggetti*, così chiamati perché hanno un'esistenza più o meno indipendente all'interno del programma e perché hanno scopi specifici.

I diversi paradigmi non limitano necessariamente il tipo di compito che può essere eseguito da un programma. Il codice dell'esempio precedente può essere riscritto secondo il paradigma OOP usando il linguaggio C++:

```
#include <iostream>

class Cube {
    float side;
public:
    Cube(float s){ side = s; }
    float volume() { return side * side * side; }
};

int main(){
    float side = 2;
    Cube cube(side);
    std::cout << "Volume: " << cube.volume() << std::endl;
    return 0;
}
```

La funzione `main()` è ancora presente. Ma ora c'è una nuova parola, `class`, che introduce la definizione di un oggetto. La classe definita, chiamata `Cube`, contiene le proprie variabili e subroutine. In OOP, una variabile è anche chiamata *attributo* e una subroutine è chiamata *metodo*.

Va oltre lo scopo di questo capitolo spiegare tutto il codice C++ dell'esempio. Ciò che è importante per noi qui è che `Cube` contiene l'attributo `side` e due metodi. Il metodo `volume()` calcola il volume del cubo.

È possibile creare diversi oggetti indipendenti dalla stessa classe, e le classi possono essere composte da altre classi.

Tenete a mente che queste stesse caratteristiche possono essere scritte in modo diverso e che gli esempi in questo capitolo sono eccessivamente semplificati. C e C++ hanno caratteristiche molto più

sofisticate che permettono costruzioni molto più complesse e pratiche.

La maggior parte dei linguaggi di programmazione non impone rigorosamente un paradigma, ma permette ai programmatore di scegliere vari aspetti di un paradigma o di un altro. JavaScript, per esempio, incorpora aspetti di diversi paradigmi. Il programmatore può scomporre l'intero programma in funzioni che non condividono uno stato comune tra loro:

```
function cube(side){  
    return side*side*side;  
}  
  
console.log("Volume: " + cube(2));
```

Anche se questo esempio è simile alla programmazione procedurale, si noti che la funzione riceve una copia di tutte le informazioni necessarie per la sua esecuzione e produce sempre lo stesso risultato per lo stesso parametro, indipendentemente dai cambiamenti che avvengono al di fuori dello scopo della funzione. Questo paradigma, chiamato *funzionale*, è fortemente influenzato dal formalismo matematico, dove ogni operazione è autosufficiente.

Un altro paradigma riguarda i linguaggi *dichiarativi*, che descrivono gli stati in cui si vuole che il sistema sia. Un linguaggio dichiarativo può capire come raggiungere gli stati specificati. SQL, il linguaggio universale per interrogare i database, è talvolta chiamato un linguaggio dichiarativo, anche se in realtà occupa una nicchia a se stante nel pantheon della programmazione.

Non esiste un paradigma universale che possa essere adottato in qualsiasi contesto. La scelta del linguaggio può anche essere limitata da quali linguaggi sono supportati sulla piattaforma o dall'ambiente di esecuzione in cui il programma sarà eseguito.

Un'applicazione web che sarà usata dal browser, per esempio, dovrà essere scritta in JavaScript, che è un linguaggio universalmente supportato dai browser. (Alcuni altri linguaggi possono essere usati perché forniscono convertitori per creare JavaScript). Quindi per il browser web - a volte chiamato *client side* o *front end* dell'applicazione web - lo sviluppatore dovrà usare i paradigmi consentiti in JavaScript. Il lato server o *back end* dell'applicazione, che gestisce le richieste dal browser, è normalmente programmato in un linguaggio diverso; PHP è il più popolare per questo scopo.

Indipendentemente dal paradigma, ogni linguaggio ha *library* precostituite di funzioni che possono essere incorporate nel codice. Le funzioni matematiche - come quelle illustrate nel codice di esempio - non hanno bisogno di essere implementate da zero, poiché il linguaggio ha già la funzione pronta all'uso. JavaScript, per esempio, fornisce l'oggetto `Math` con le operazioni matematiche più comuni.

Anche funzioni più specializzate sono di solito disponibili dal fornitore del linguaggio o da

sviluppatori di terze parti. Queste librerie di risorse extra possono essere in forma di codice sorgente; cioè, in file extra che sono incorporati nel file in cui saranno usati. In JavaScript, l'incorporazione è fatta con `import from`:

```
import { OrbitControls } from 'modules/OrbitControls.js';
```

Questo tipo di importazione, in cui la risorsa incorporata è anche un file di codice sorgente, è usato più spesso nei cosiddetti *linguaggi interpretati*. I *linguaggi compilati* permettono, tra le altre cose, l'incorporazione di caratteristiche precompilate nel linguaggio macchina, cioè le *librerie compilate*. La prossima sezione spiega le differenze tra questi tipi di linguaggi.

Compilatori e Interpreti

Come già sappiamo, il codice sorgente è una rappresentazione simbolica di un programma che deve essere tradotto in linguaggio macchina per essere eseguito.

Grosso modo, ci sono due modi possibili di fare la traduzione: convertire il codice sorgente in anticipo per l'esecuzione futura, o convertire il codice al momento della sua esecuzione. I linguaggi della prima modalità sono chiamati *linguaggi compilati* e i linguaggi della seconda modalità sono chiamati *linguaggi interpretati*. Alcuni linguaggi interpretati prevedono la compilazione come opzione, in modo che il programma possa eseguirsi più velocemente.

Nei linguaggi compilati, c'è una chiara distinzione tra il codice sorgente del programma e il programma stesso, che sarà eseguito dal computer. Una volta compilato, il programma di solito funziona solo sul sistema operativo e sulla piattaforma per cui è stato compilato.

In un linguaggio interpretato, il codice sorgente stesso è trattato come il programma, e il processo di conversione in linguaggio macchina è trasparente al programmatore. Per un linguaggio interpretato è comune chiamare il codice sorgente uno *script*. L'interprete traduce lo script nel linguaggio macchina per il sistema su cui viene eseguito.

Compilazione e Compilatori

Il linguaggio di programmazione C è uno dei più noti esempi di linguaggio compilato. I maggiori punti di forza del linguaggio C sono la sua flessibilità e le sue prestazioni. Sia i supercomputer ad alte prestazioni sia i microcontrollori negli elettrodomestici possono essere programmati in linguaggio C. Altri esempi di linguaggi compilati popolari sono C++ e C# (C sharp). Come suggeriscono i loro nomi, questi linguaggi sono ispirati al C, ma includono caratteristiche che supportano il paradigma orientato agli oggetti.

Lo stesso programma scritto in C o C++ può essere compilato per diverse piattaforme, richiedendo

poco o nessun cambiamento al codice sorgente. È il compilatore che definisce la piattaforma di destinazione del programma. Ci sono compilatori specifici per piattaforma così come compilatori multipiattaforma come GCC (che sta per *GNU Compiler Collection*) che possono produrre programmi binari per molte architetture diverse.

NOTE

Ci sono anche strumenti che automatizzano il processo di compilazione. Invece di invocare direttamente il compilatore, il programmatore crea un file che indica i diversi passi di compilazione da eseguire automaticamente. Lo strumento tradizionale usato per questo scopo è `make`, ma un certo numero di strumenti più recenti come *Maven* e *Gradle* sono diffusamente utilizzati. L'intero processo di compilazione è automatizzato quando si usa un IDE.

Il processo di compilazione non sempre genera un programma binario in linguaggio macchina. Ci sono linguaggi compilati che producono un programma in un formato genericamente chiamato *bytecode*. Come uno script, il bytecode non è in un linguaggio specifico della piattaforma, quindi richiede un programma interprete che lo traduca in linguaggio macchina. In questo caso, il programma interprete è chiamato semplicemente *runtime*.

Il linguaggio Java adotta questo approccio, così i programmi compilati scritti in Java possono essere usati su diversi sistemi operativi. Nonostante il suo nome, Java *non è* collegato a JavaScript.

Il *bytecode* è più vicino al linguaggio macchina che al codice sorgente, quindi la sua esecuzione tende a essere relativamente più veloce. Poiché c'è ancora un processo di conversione durante l'esecuzione del bytecode, è difficile ottenere le stesse prestazioni di un programma equivalente compilato in linguaggio macchina.

Interpretazione e Interpreti

Nei linguaggi interpretati come JavaScript, Python e PHP il programma non ha bisogno di essere precompilato, rendendo più facile lo sviluppo e la modifica. Invece di compilarlo, lo script viene eseguito da un altro programma chiamato *interprete*. Di solito, l'interprete di un linguaggio prende il nome dal linguaggio stesso. L'interprete di uno script Python, per esempio, è un programma chiamato `python`. L'interprete JavaScript è molto spesso il browser web, ma gli script possono anche essere eseguiti dal programma `node.js` al di fuori di un browser. Poiché viene convertito in istruzioni binarie ogni volta che viene eseguito, un programma in linguaggio interpretato tende a essere più lento di un equivalente in linguaggio compilato.

Nulla impedisce che la stessa applicazione abbia componenti scritte in linguaggi diversi. Se necessario, questi componenti possono comunicare attraverso un'interfaccia di programmazione dell'applicazione (API) reciprocamente comprensibile.

Il linguaggio Python, per esempio, ha capacità molto sofisticate di *data mining* e tabulazione dei dati. Lo sviluppatore può scegliere Python per scrivere le parti del programma che si occupano di questi aspetti e un altro linguaggio, come C++, per eseguire l'elaborazione numerica più pesante. È possibile adottare questa strategia anche quando non ci sono API che permettono la comunicazione diretta tra i due componenti. Il codice scritto in Python può per esempio generare un file nel formato appropriato per essere usato da un programma scritto in C++.

Anche se è possibile scrivere quasi qualsiasi programma in qualsiasi linguaggio, lo sviluppatore dovrebbe adottare quello che è più in linea con lo scopo dell'applicazione. Così facendo, si beneficia del riutilizzo di componenti già testati e ben documentati.

Esercizi Guidati

1. Che tipo di programma può essere usato per modificare il codice sorgente?

2. Che tipo di strumento aiuta a integrare il lavoro di diversi sviluppatori nella stessa base di codice?

Esercizi Esplorativi

1. Supponiamo che vogliate scrivere un gioco in 3D da eseguire nel browser. Le applicazioni web e i giochi sono programmati in JavaScript. Anche se è possibile scrivere tutte le funzioni grafiche da zero, è più produttivo usare una libreria già pronta per questo scopo. Quali librerie di terze parti forniscono capacità per l'animazione 3D in JavaScript?

2. Oltre a PHP, quali altri linguaggi possono essere usati sul lato server di un'applicazione web?

Sommario

Questa lezione tratta i concetti più essenziali dello sviluppo del software. Lo sviluppatore deve essere consapevole dei principali linguaggi di programmazione e del corretto scenario d'uso di ciascuno. Questa lezione affronta i seguenti concetti e procedure:

- Cos'è il codice sorgente.
- Editor di codice sorgente e strumenti correlati.
- Paradigmi di programmazione procedurale, orientata agli oggetti, funzionale e dichiarativa.
- Caratteristiche dei linguaggi compilati e interpretati.

Risposte agli Esercizi Guidati

1. Che tipo di programma può essere usato per modificare il codice sorgente?

In linea di principio, qualsiasi programma in grado di modificare il testo.

2. Che tipo di strumento aiuta a integrare il lavoro di diversi sviluppatori nella stessa base di codice?

Un sistema di controllo dei sorgenti o delle versioni, come Git.

Risposte agli Esercizi Esplorativi

1. Supponiamo che vogliate scrivere un gioco in 3D da eseguire nel browser. Le applicazioni web e i giochi sono programmati in JavaScript. Anche se è possibile scrivere tutte le funzioni grafiche da zero, è più produttivo usare una libreria già pronta per questo scopo. Quali librerie di terze parti forniscono capacità per l'animazione 3D in JavaScript?

Ci sono molte opzioni per librerie di grafica 3D per JavaScript, come *three.js* e *BabylonJS*.

2. Oltre a PHP, quali altri linguaggi possono essere usati sul lato server di un'applicazione web?

Qualsiasi linguaggio supportato dall'applicazione server HTTP usata sull'host del server. Alcuni esempi sono Python, Ruby, Perl e lo stesso JavaScript.



031.2 Architettura di un'Applicazione Web

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 031.2

Peso

2

Arearie di Conoscenza Chiave

- Comprendere il principio dell'informatica client e server
- Comprendere il ruolo dei browser web ed conoscere i browser web comunemente usati
- Comprendere il ruolo dei server web e dei server di applicazioni
- Comprendere le tecnologie e gli standard di sviluppo web comuni
- Comprendere i principi delle API
- Comprendere i fondamenti dei database relazionali e non relazionali (NoSQL)
- Conoscenza dei sistemi di gestione di database Open Source comunemente usati
- Conoscenza di REST e GraphQL
- Conoscenza delle applicazioni a pagina singola
- Conoscenza del packaging delle applicazioni web
- Conoscenza di WebAssembly
- Conoscenza dei sistemi di gestione dei contenuti

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- Chrome, Edge, Firefox, Safari, Internet Explorer

- HTML, CSS, JavaScript
- SQLite, MySQL, MariaDB, PostgreSQL
- MongoDB, CouchDB, Redis



031.2 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	031 Sviluppo Software e Tecnologie Web
Obiettivo:	031.2 Architettura di un'Applicazione Web
Lezione:	1 di 1

Introduzione

La parola *applicazione* ha un ampio significato nel gergo tecnologico. Quando l'applicazione è un programma tradizionale, eseguito localmente e autosufficiente nel suo scopo, sia l'interfaccia operativa dell'applicazione sia le componenti di elaborazione dati sono integrate in un unico "pacchetto". Una *web application* è diversa perché adotta il modello client/server e la sua parte client è basata su HTML, che è ottenuto dal server e, in generale, reso attraverso un browser.

Client e Server

Nel modello client/server, parte del lavoro viene fatto localmente sul *lato client* e parte del lavoro viene fatto in remoto, sul *lato server*. Quali compiti vengono eseguiti da ciascuna parte varia a seconda dello scopo dell'applicazione, ma in generale spetta al client fornire un'interfaccia all'utente e impaginare il contenuto in modo attraente. Spetta al server eseguire la parte elaborativa dell'applicazione, elaborando e rispondendo alle richieste fatte dal client. In un'applicazione di

shopping, per esempio, l'applicazione client visualizza un'interfaccia per l'utente per scegliere e pagare i prodotti, ma la fonte dei dati e i record delle transazioni sono tenuti sul server remoto, a cui si accede tramite la rete. Le applicazioni web eseguono questa comunicazione su Internet, di solito tramite il protocollo HTTP (Hypertext Transfer Protocol).

Una volta caricato dal browser, il lato client dell'applicazione inizia l'interazione con il server quando necessario o conveniente. I server di applicazioni web offrono un'interfaccia di programmazione di applicazioni (API) che definisce le richieste disponibili e il modo in cui queste richieste devono essere fatte. Così, il client costruisce una richiesta nel formato definito dall'API e la invia al server, che controlla i prerequisiti per la richiesta e rimanda la risposta appropriata.

Mentre il client, sotto forma di applicazione *mobile* o *browser desktop*, è un programma autonomo per quanto riguarda l'interfaccia utente e le istruzioni per comunicare con il server, il browser deve ottenere la pagina HTML e i componenti associati—come immagini, CSS e JavaScript—che definiscono l'interfaccia e le istruzioni per comunicare con il server.

I linguaggi di programmazione e le piattaforme usate da client e server sono indipendenti, ma usano un protocollo di comunicazione reciprocamente comprensibile. La parte server è quasi sempre eseguita da un programma senza interfaccia grafica, che gira in ambienti informatici altamente disponibili in modo che sia sempre pronto a rispondere alle richieste. Al contrario, la parte client viene eseguita su un qualsiasi dispositivo che è in grado di mostrare un'interfaccia HTML, come smartphone, pc e cos' via.

Oltre a essere essenziale per certi scopi, l'adozione del modello client/server permette a un'applicazione di ottimizzare diversi aspetti dello sviluppo e della manutenzione, poiché ogni parte può essere progettata per il suo scopo specifico. Un'applicazione che visualizza mappe e percorsi, per esempio, non ha bisogno di avere tutte le mappe memorizzate localmente. Solo le mappe relative alla posizione di interesse dell'utente sono richieste, quindi solo quelle mappe sono richieste dal server centrale.

Gli sviluppatori hanno il controllo diretto sul server, quindi possono anche modificare il client che viene fornito da esso. Questo permette agli sviluppatori di migliorare l'applicazione, in misura maggiore o minore, senza che l'utente debba esplicitamente installare nuove versioni.

La Parte Client

Un'applicazione web dovrebbe funzionare allo stesso modo su uno qualsiasi dei browser più popolari, a patto che il browser sia aggiornato. Alcuni browser possono essere incompatibili con le recenti innovazioni, ma solo le applicazioni sperimentali utilizzano caratteristiche non ancora ampiamente adottate.

I problemi di incompatibilità erano più comuni in passato, quando i diversi browser avevano il proprio *motore di rendering* e c'era meno cooperazione nel formulare e adottare standard. Il motore di rendering è il componente principale del browser, in quanto è responsabile della trasformazione dell'HTML e di altri componenti associati negli elementi visivi e interattivi dell'interfaccia. Alcuni browser, in particolare Internet Explorer, avevano bisogno di un trattamento speciale nel codice per non compromettere il funzionamento previsto delle pagine.

Oggi ci sono differenze minime tra i principali browser e le incompatibilità sono rare. I browser Chrome ed Edge usano lo stesso motore di rendering (chiamato Blink). Il browser Safari, e altri browser offerti sull'*App Store* di iOS, usano il motore WebKit. Firefox usa un motore chiamato Gecko. Questi tre motori rappresentano praticamente tutti i browser usati oggi. Anche se sviluppati separatamente, i tre motori sono progetti *open source* e c'è cooperazione tra gli sviluppatori, il che facilita la compatibilità, la manutenzione e l'adozione di standard.

Poiché gli sviluppatori di browser hanno fatto tanti sforzi per rimanere compatibili, il server non è normalmente legato a un solo tipo di client. In linea di principio, un server HTTP può comunicare con qualsiasi client che sia anche in grado di comunicare via HTTP. In un'applicazione di mappe, per esempio, il client può essere un'applicazione mobile o un browser che carica l'interfaccia HTML dal server.

Tipi di Client Web

Ci sono applicazioni *mobile* e *desktop* la cui interfaccia è resa tramite HTML e, come i browser, possono usare JavaScript come linguaggio di programmazione. Tuttavia, a differenza del client caricato nel browser, l'HTML e i componenti necessari al funzionamento del client nativo sono presenti localmente dall'installazione dell'applicazione. Un'applicazione che funziona in questo modo è virtualmente identica a una pagina HTML (è persino probabile che entrambe siano rese dallo stesso motore). Ci sono anche le *progressive web apps* (PWA), che permettono di "impacchettare" i client delle applicazioni web per l'uso offline - limitatamente alle funzioni che non richiedono una comunicazione immediata con il server. Per quanto riguarda ciò che l'applicazione può fare, non c'è differenza tra l'esecuzione nel browser o confezionata in una PWA, tranne che in quest'ultima lo sviluppatore ha più controllo su ciò che viene memorizzato localmente.

Il *rendering* delle interfacce HTML è un'attività così ricorrente che il motore è di solito un componente software separato, presente nel sistema operativo. La sua presenza come componente separato permette ad applicazioni diverse di incorporarlo senza doverlo incorporare nel pacchetto dell'applicazione. Questo modello delega anche la manutenzione del motore di rendering al sistema operativo, facilitando gli aggiornamenti. È molto importante mantenere aggiornato un componente così cruciale per evitare possibili problematiche.

Indipendentemente dal loro metodo di consegna, le applicazioni scritte in HTML girano su un livello

di astrazione creato dal motore, che funziona come un ambiente di esecuzione isolato. In particolare, nel caso di un client che gira sul browser, l'applicazione ha a disposizione solo le risorse offerte dal browser. Le funzionalità di base, come l'interazione con gli elementi della pagina e la richiesta di file tramite HTTP, sono sempre disponibili. Le risorse che possono contenere informazioni sensibili, come l'accesso ai file locali, la posizione geografica, la fotocamera e il microfono, richiedono l'autorizzazione esplicita dell'utente prima che l'applicazione sia in grado di utilizzarle.

Linguaggi di un Client Web

L'elemento centrale di un client di un'applicazione web che gira sul server è il documento HTML. Oltre a presentare gli elementi dell'interfaccia che il browser visualizza in modo strutturato, il documento HTML contiene gli indirizzi di tutti i file necessari per la corretta presentazione e funzionamento del client.

L'HTML da solo non ha molta versatilità per costruire interfacce più elaborate e non ha caratteristiche di programmazione generale. Per questo motivo, un documento HTML che deve funzionare come applicazione client è sempre accompagnato da uno o più set di CSS e JavaScript.

Il CSS (*Cascading Style Sheets*) può essere fornito come file separato o direttamente nel file HTML stesso. Lo scopo principale dei CSS è quello di regolare l'aspetto e la disposizione degli elementi dell'interfaccia HTML. Anche se non è strettamente necessario, le interfacce più sofisticate di solito richiedono modifiche alle proprietà CSS degli elementi per soddisfare le loro esigenze.

JavaScript è un componente praticamente indispensabile. Le procedure scritte in JavaScript rispondono a eventi nel browser. Questi eventi possono essere causati dall'utente e non. Senza JavaScript, un documento HTML è praticamente limitato al testo e alle immagini. L'uso di JavaScript nei documenti HTML permette di estendere l'interattività oltre i collegamenti ipertestuali e i moduli, rendendo la pagina visualizzata dal browser come un'interfaccia di applicazione convenzionale.

JavaScript è un linguaggio di programmazione genico, ma il suo uso principale è nelle applicazioni web. Le caratteristiche dell'ambiente di esecuzione del browser sono accessibili attraverso parole chiave JavaScript, utilizzate in uno script per eseguire l'operazione desiderata. Il termine `document`, per esempio, è usato nel codice JavaScript per riferirsi al documento HTML associato al codice JavaScript. Nel contesto del linguaggio JavaScript, `document` è un oggetto globale con proprietà e metodi che possono essere usati per ottenere informazioni da qualsiasi elemento del documento HTML. Ancora più importante, si può usare l'oggetto `document` per modificare i suoi elementi e associarli ad azioni personalizzate scritte in JavaScript.

Un'applicazione client basata su tecnologie web è *multipiattaforma*, perché può essere eseguita su qualsiasi dispositivo che abbia un browser web compatibile.

Essere confinati al browser, tuttavia, impone limitazioni alle applicazioni web rispetto alle

applicazioni native. L’intermediazione effettuata dal browser permette una programmazione di livello superiore e aumenta la sicurezza, ma aumenta anche l’elaborazione e il consumo di memoria.

Gli sviluppatori lavorano continuamente sui browser per fornire più funzioni e migliorare le prestazioni delle applicazioni JavaScript, ma ci sono aspetti intrinseci all’esecuzione di script come JavaScript che impongono uno svantaggio rispetto ai programmi nativi per lo stesso hardware.

Una caratteristica che migliora significativamente le prestazioni delle applicazioni JavaScript in esecuzione sul browser è *WebAssembly*. WebAssembly è un tipo di JavaScript compilato che produce codice sorgente scritto in un linguaggio più efficiente e di livello inferiore, come il linguaggio C. WebAssembly può accelerare principalmente le attività ad alta intensità di processore, perché evita gran parte della traduzione eseguita dal browser quando si esegue un programma scritto in JavaScript convenzionale.

Indipendentemente dai dettagli di implementazione dell’applicazione, tutto il codice HTML, CSS, JavaScript e i file multimediali devono prima essere ottenuti dal server. Il browser ottiene questi file proprio come una pagina Internet, cioè con un indirizzo a cui il browser accede.

Una pagina web che funge da interfaccia per un’applicazione web è come un semplice documento HTML, ma aggiunge ulteriori comportamenti. Nelle pagine convenzionali l’utente viene indirizzato a un’altra pagina dopo aver cliccato su un link. Le applicazioni web possono presentare la loro interfaccia e rispondere agli eventi dell’utente senza caricare nuove pagine nella finestra del browser. La modifica di questo comportamento standard nelle pagine HTML viene fatta tramite la programmazione JavaScript.

Un client webmail, per esempio, visualizza i messaggi e passa da una cartella all’altra senza lasciare la pagina. Questo è possibile perché il client usa JavaScript per reagire alle azioni dell’utente e fare richieste appropriate al server. Se l’utente clicca sull’oggetto di un messaggio nella posta in arrivo, un codice JavaScript associato a questo evento richiede il contenuto di quel messaggio al server (usando la chiamata API corrispondente). Non appena il client riceve la risposta, il browser visualizza il messaggio nella porzione appropriata della stessa pagina. Diversi client webmail possono adottare strategie diverse, ma tutti usano questo stesso principio.

Quindi, oltre a fornire i file che compongono il client al browser, il server deve anche essere in grado di gestire richieste come quella del client webmail, quando chiede il contenuto di un messaggio specifico. Ogni richiesta che il client può fare ha una procedura predefinita per rispondere sul server, le cui API possono definire diversi metodi per identificare a quale procedura si riferisce la richiesta. I metodi più comuni sono:

- Indirizzi, attraverso uno Uniform Resource Locator (URL)
- Campi nelle intestazioni HTTP

- metodi GET/POST
- WebSocket

Un metodo può essere più adatto di un altro, a seconda dello scopo della richiesta e di altri criteri presi in considerazione dallo sviluppatore. In generale, le applicazioni web utilizzano una combinazione di metodi, ognuno in una circostanza specifica.

Il paradigma *Representational State Transfer* (REST) è ampiamente utilizzato per la comunicazione nelle applicazioni web, perché si basa sui metodi di base disponibili in HTTP. L'intestazione di una richiesta HTTP inizia con una parola chiave che definisce l'operazione di base da eseguire: GET, POST, PUT, DELETE, ecc., accompagnata da un URL corrispondente dove l'azione sarà applicata. Se l'applicazione richiede operazioni più specifiche, con una descrizione più dettagliata dell'operazione richiesta, il protocollo GraphQL può essere una scelta più appropriata.

Le applicazioni sviluppate utilizzando il modello client/server sono soggette a instabilità nella comunicazione. Per questo motivo, l'applicazione client deve sempre adottare strategie di trasferimento dati efficienti per favorire la sua coerenza e non danneggiare l'esperienza dell'utente.

Il Lato Server

Nonostante sia l'attore principale di un'applicazione web, il server è il lato passivo della comunicazione, rispondendo solo alle richieste fatte dal client. Nel gergo web, *server* può riferirsi alla macchina che riceve le richieste, al programma che gestisce specificamente le richieste HTTP, o allo script destinatario che produce una risposta alla richiesta. Quest'ultima definizione è la più rilevante nel contesto dell'architettura delle applicazioni web, ma sono tutte strettamente correlate. Anche se sono solo parzialmente nell'ambito dello sviluppatore di applicazioni server, la macchina, il sistema operativo e il server HTTP non possono essere ignorati, perché sono fondamentali per il funzionamento del server di applicazioni e spesso si intersecano.

Gestire i Percorsi dalle Richieste

I server HTTP, come Apache e NGINX, di solito richiedono modifiche di configurazione specifiche per soddisfare le esigenze dell'applicazione. Per impostazione predefinita, i server HTTP tradizionali associano direttamente il percorso indicato nella richiesta a un file sul file system locale. Se il server HTTP di un sito web mantiene i suoi file HTML nella directory /srv/www, per esempio, una richiesta con il percorso /it/about.html riceverà come risposta il contenuto del file /srv/www/it/about.html, se il file esiste. Siti web più sofisticati, e specialmente applicazioni web, richiedono trattamenti personalizzati per diversi tipi di richieste. In questo scenario, parte dell'implementazione dell'applicazione consiste nel modificare le impostazioni del server HTTP per soddisfare i requisiti dell'applicazione.

In alternativa ci sono *framework* che permettono di integrare la gestione delle richieste HTTP e l'implementazione del codice dell'applicazione in un unico luogo, permettendo allo sviluppatore di concentrarsi più sullo scopo dell'applicazione che sui dettagli della piattaforma. In *Node.js Express*, per esempio, tutta la mappatura delle richieste e la programmazione corrispondente sono implementate utilizzando JavaScript. Dato che la programmazione dei client è di solito fatta in JavaScript, molti sviluppatori considerano una buona idea dal punto di vista della manutenzione del codice usare lo stesso linguaggio per client e server. Altri linguaggi comunemente usati per implementare la parte server, sia nei framework che nei tradizionali server HTTP, sono PHP, Python, Ruby, Java e C#.

Sistemi di Gestione di Database

Sta alla discrezione del team di sviluppo come i dati ricevuti o richiesti dal client vengono memorizzati sul server, ma ci sono linee guida generali che si applicano alla maggior parte dei casi. È conveniente mantenere il contenuto statico - immagini, codice JavaScript e CSS che non cambiano a breve termine - come file convenzionali, o sul file system del server o distribuito attraverso una *content delivery network* (CDN). Altri tipi di contenuto, come i messaggi di posta elettronica in un'applicazione webmail, i dettagli dei prodotti in un'applicazione di shopping, e i log delle transazioni, sono più convenientemente memorizzati in un *database management system* (DBMS).

Il tipo più tradizionale di sistema di gestione di database è la *base di dati relazionale*. In esso, il progettista dell'applicazione definisce le tabelle di dati e il formato di input accettato da ogni tabella. L'insieme delle tabelle nel database contiene tutti i dati dinamici consumati e prodotti dall'applicazione. Un'applicazione di shopping, per esempio, può avere una tabella che contiene una voce con i dettagli di ogni prodotto nel negozio e una tabella che registra gli articoli acquistati da un utente. La tabella degli articoli acquistati contiene riferimenti alle voci della tabella dei prodotti, creando relazioni tra le tabelle. Questo approccio può ottimizzare l'immagazzinamento e l'accesso ai dati, oltre a permettere interrogazioni in tabelle combinate usando il linguaggio adottato dal sistema di gestione del database. Il linguaggio di database relazionale più popolare è lo *Structured Query Language* (SQL, pronunciato "sequel"), adottato dai database open source *SQLite*, *MySQL*, *MariaDB* e *PostgreSQL*.

Un'alternativa ai database relazionali è una forma di database che non richiede una struttura rigida per i dati. Questi database sono chiamati *database non relazionali* o semplicemente *NoSQL*. Anche se possono incorporare alcune caratteristiche simili a quelle che si trovano nei database relazionali, l'obiettivo è quello di consentire una maggiore flessibilità nella memorizzazione e l'accesso ai dati memorizzati, passando il compito di elaborare tali dati all'applicazione stessa. *MongoDB*, *CouchDB* e *Redis* sono comuni sistemi di gestione di database non relazionali.

Gestione del Contenuto

Indipendentemente dal modello di database adottato, le applicazioni devono aggiungere dati e probabilmente aggiornarli nel corso della loro vita. In alcune applicazioni, come la webmail, gli utenti stessi forniscono dati al database quando usano il client per inviare e ricevere messaggi. In altri casi, come nell'applicazione di shopping, è importante permettere ai manutentori dell'applicazione di modificare il database senza dover ricorrere alla programmazione. Molte organizzazioni adottano quindi un qualche tipo di *content management system* (CMS), che permette agli utenti non tecnici di amministrare l'applicazione. Pertanto, per la maggior parte delle applicazioni web, è necessario implementare almeno due tipi di client: un client non privilegiato, utilizzato dagli utenti ordinari, e client privilegiati, utilizzati da utenti speciali per mantenere e aggiornare le informazioni presentate dall'applicazione.

Esercizi Guidati

1. Quale linguaggio di programmazione viene usato insieme all'HTML per creare applicazioni web client?

2. In che modo un'applicazione web differisce da un'applicazione nativa?

3. In che modo un'applicazione web differisce da un'applicazione nativa nell'accesso all'hardware locale?

4. Citare una caratteristica del client di un'applicazione web che lo distingue da una normale pagina web.

Esercizi Esplorativi

1. Quale caratteristica offrono i browser moderni per superare le scarse prestazioni dei client delle applicazioni web ad alta intensità di CPU?

2. Se un'applicazione web utilizza il paradigma REST per la comunicazione client/server, quale metodo HTTP dovrebbe essere utilizzato quando il client richiede al server di cancellare una risorsa specifica?

3. Citare cinque linguaggi di scripting per server supportati dal server HTTP Apache.

4. Perché i database non relazionali sono considerati più facili da mantenere e aggiornare dei database relazionali?

Sommario

Questa lezione tratta i concetti e gli standard della tecnologia e dell'architettura dello sviluppo web. Il principio è semplice: il browser web esegue l'applicazione client, che comunica con l'applicazione principale in esecuzione nel server. Anche se semplice nel principio, le applicazioni web devono combinare molte tecnologie per adottare il principio del calcolo client e server sul web. La lezione passa attraverso i seguenti concetti:

- Il ruolo dei browser e dei server web.
- Tecnologie e standard comuni di sviluppo web.
- Come i client web possono comunicare con il server.
- Tipi di server web e server di database.

Risposte agli Esercizi Guidati

- Quale linguaggio di programmazione viene usato insieme all'HTML per creare applicazioni web client?

JavaScript.

- In che modo un'applicazione web differisce da un'applicazione nativa?

Un'applicazione web non viene installata. Invece, parti di essa vengono eseguite sul server e l'interfaccia client viene eseguita in un normale browser web.

- In che modo un'applicazione web differisce da un'applicazione nativa nell'accesso all'hardware locale?

Tutti gli accessi alle risorse locali, come lo *storage*, le telecamere o i microfoni, sono mediati dal browser e richiedono l'autorizzazione esplicita dell'utente per funzionare.

- Citare una caratteristica del client di un'applicazione web che lo distingue da una normale pagina web.

L'interazione con le pagine web tradizionali è fondamentalmente limitata a collegamenti ipertestuali e all'invio di moduli, mentre i client delle applicazioni web sono più vicini a un'interfaccia di applicazione convenzionale.

Risposte agli Esercizi Esplorativi

- Quale caratteristica offrono i browser moderni per superare le scarse prestazioni dei client delle applicazioni web ad alta intensità di CPU?

Gli sviluppatori possono usare WebAssembly per implementare le parti ad alta intensità di CPU dell'applicazione client. Il codice WebAssembly generalmente ha prestazioni migliori del tradizionale JavaScript, perché richiede meno traduzione di istruzioni.

- Se un'applicazione web utilizza il paradigma REST per la comunicazione client/server, quale metodo HTTP dovrebbe essere utilizzato quando il client richiede al server di cancellare una risorsa specifica?

REST si basa su metodi HTTP standard, quindi in questo caso si dovrebbe usare il metodo standard DELETE.

- Citare cinque linguaggi di scripting per server supportati dal server HTTP Apache.

PHP, Go, Perl, Python, e Ruby.

- Perché i database non relazionali sono considerati più facili da mantenere e aggiornare dei database relazionali?

A differenza dei database relazionali, i database non relazionali non richiedono che i dati si adattino a strutture rigide predefinite, rendendo più facile implementare cambiamenti nelle strutture dei dati senza influenzare i dati esistenti.



031.3 Nozioni Base sull'HTTP

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 031.3

Peso

3

Arete di Conoscenza Chiave

- Comprendere i metodi HTTP GET e POST, i codici di stato, le intestazioni e i tipi di contenuto.
- Comprendere la differenza tra contenuto statico e dinamico
- Comprendere le URL HTTP
- Comprendere come le URL HTTP sono mappate nei percorsi del file system
- Caricare file nella root dei documenti di un server web
- Comprendere il caching
- Comprendere i cookie
- Conoscenza delle sessioni e il dirottamento delle sessioni
- Conoscenza dei server HTTP comunemente usati
- Conoscenza di HTTPS e TLS
- Conoscenza dei web socket
- Conoscenza degli host virtuali
- Conoscenza dei server HTTP comuni
- Conoscenza dei requisiti e delle limitazioni della larghezza di banda e della latenza della rete

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- GET, POST
- 200, 301, 302, 401, 403, 404, 500
- Apache HTTP Server (httpd), NGINX



031.3 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	031 Sviluppo Software e Tecnologie Web
Obiettivo:	031.3 Nozioni Base sull'HTTP
Lezione:	1 di 1

Introduzione

L'*HyperText Transfer Protocol* (HTTP) definisce come un client chiede al server una risorsa specifica. Il suo principio di funzionamento è abbastanza semplice: il client crea un messaggio di richiesta che identifica la risorsa di cui ha bisogno e lo inoltra al server attraverso la rete. A sua volta, il server HTTP valuta dove estrarre la risorsa richiesta e invia un messaggio di risposta al client. Il messaggio di risposta contiene dettagli sulla risorsa richiesta, seguito dalla risorsa stessa.

Più precisamente, HTTP è l'insieme di regole che definiscono come l'applicazione client deve formattare i messaggi di *request* che saranno inviati al server. Il server segue quindi le regole HTTP per interpretare la richiesta e formattare i messaggi di *reply*. Oltre a richiedere o trasferire il contenuto richiesto, i messaggi HTTP contengono informazioni extra sul client e sul server coinvolti, sul contenuto stesso, e anche sulla sua indisponibilità. Se una risorsa non può essere inviata, un codice nella risposta spiega il motivo dell'indisponibilità e, se possibile, indica dove la risorsa è stata spostata.

La parte del messaggio che definisce i dettagli della risorsa e altre informazioni di contesto è chiamata *header* del messaggio. La parte che segue l'intestazione, che contiene il contenuto della risorsa corrispondente, è chiamata il *payload* del messaggio. Sia i messaggi di richiesta sia quelli di

risposta possono avere un payload, ma nella maggior parte dei casi, solo il messaggio di risposta ne ha uno.

La Richiesta del Client

Il primo stadio di uno scambio di dati HTTP tra il client e il server è iniziato dal client, quando scrive un messaggio di richiesta al server. Prendiamo, per esempio, un compito comune del browser: caricare una pagina HTML da un server che ospita un sito web, come <https://learning.lpi.org/>. L'indirizzo, o URL, fornisce diverse informazioni rilevanti. Tre informazioni appaiono in questo particolare esempio:

- Il protocollo: HyperText Transfer Protocol Secure ([https](https://)), una versione criptata di HTTP.
- Il nome di rete dell'host web (learning.lpi.org).
- La posizione della risorsa richiesta sul server (la directory </en/> - in questo caso, la versione inglese della home page).

NOTE

Un *Uniform Resource Locator* (URL) è un indirizzo che punta a una risorsa su Internet. Questa risorsa è di solito un file che può essere ottenuto da un server remoto, ma gli URL possono anche indicare contenuti e flussi di dati generati dinamicamente.

Come il client gestisce l'URL

Prima di contattare il server, il client deve convertire learning.lpi.org nel suo indirizzo IP corrispondente. Il client usa un altro servizio Internet, il *Domain Name System* (DNS), per richiedere l'indirizzo IP di un nome host da uno o più server DNS predefiniti (i server DNS sono solitamente definiti automaticamente dall'Internet Service Provider, ISP).

Con l'indirizzo IP del server, il client cerca di connettersi alla porta HTTP o HTTPS. Le porte di rete sono numeri di identificazione definiti dal *Transmission Control Protocol* (TCP) per interconnettere e identificare canali di comunicazione distinti all'interno di una connessione client/server. Per default, i server HTTP ricevono richieste sulle porte TCP 80 (HTTP) e 443 (HTTPS).

NOTE

Ci sono altri protocolli usati dalle applicazioni web per implementare la comunicazione client/server. Per le chiamate audio e video, per esempio, è più appropriato usare *WebSocket*, un protocollo di livello inferiore che è più efficiente di HTTP per trasferire flussi di dati in entrambe le direzioni.

Il formato del messaggio di richiesta che il client invia al server è lo stesso in HTTP e HTTPS. HTTPS è già più utilizzato di HTTP, perché tutti gli scambi di dati tra client e server sono criptati, una

caratteristica indispensabile per promuovere la privacy e la sicurezza sulle reti pubbliche. La connessione criptata viene stabilita tra il client e il server ancor prima che qualsiasi messaggio HTTP venga scambiato, utilizzando il protocollo crittografico *Transport Layer Security* (TLS). In questo modo, tutta la comunicazione HTTPS è incapsulata da TLS. Una volta decrittata, la richiesta o la risposta trasmessa su HTTPS non è diversa da una richiesta o risposta fatta esclusivamente su HTTP.

Il terzo elemento del nostro URL, `/it/`, sarà interpretato dal server come la posizione o il percorso della risorsa richiesta. Se il percorso non è fornito nell'URL, verrà utilizzata la posizione predefinita `/`. L'implementazione più semplice di un server HTTP associa i percorsi negli URL ai file presenti sul file system dove il server è in esecuzione, ma questa è solo una delle molte opzioni disponibili su server HTTP più sofisticati.

Il Messaggio di Richiesta

HTTP opera attraverso una connessione già stabilita tra client e server, di solito implementata in TCP e criptata con TLS. Infatti, una volta che una connessione che soddisfa i requisiti imposti dal server è pronta, una richiesta HTTP digitata a mano in testo semplice potrebbe generare la risposta del server. In pratica, tuttavia, i programmatori raramente hanno bisogno di implementare routine per comporre messaggi HTTP, poiché la maggior parte dei linguaggi di programmazione fornisce meccanismi che automatizzano la realizzazione del messaggio HTTP. Nel caso dell'URL di esempio, `https://learning.lpi.org/it/`, il messaggio di richiesta più semplice possibile avrebbe il seguente contenuto:

```
GET /en/ HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: text/html
```

La prima parola della prima riga identifica il *metodo* HTTP: definisce quale operazione il client vuole eseguire sul server. Il metodo GET informa il server che il client richiede la risorsa che segue: `/it/`. Sia il client sia il server possono supportare più di una versione del protocollo HTTP, quindi anche la versione da adottare nello scambio di dati viene fornita nella prima riga: HTTP/1.1.

La versione più recente del protocollo HTTP è HTTP/2. Tra le varie differenze, i messaggi scritti in HTTP/2 sono codificati in una struttura binaria, mentre i messaggi scritti in HTTP/1.1 sono inviati in testo semplice. Questo cambiamento ottimizza la velocità di trasmissione dei dati, ma il contenuto dei messaggi è fondamentalmente lo stesso.

NOTE

L'intestazione può contenere più righe dopo la prima per contestualizzare e aiutare a identificare la richiesta al server. Il campo di intestazione `Host`, per esempio, può sembrare ridondante, perché l'host del server è stato ovviamente identificato dal client per stabilire la connessione ed è ragionevole supporre che il server conosca la propria identità. Tuttavia è importante informare l'host del nome dell'host previsto nell'intestazione della richiesta, perché è pratica comune utilizzare lo stesso server HTTP per ospitare più di un sito web. (In questo scenario, ogni host specifico è chiamato *virtual host*.) Pertanto, il campo `Host` viene utilizzato dal server HTTP per identificare a quale host si riferisce la richiesta.

Il campo di intestazione `User-Agent` contiene dettagli sul programma client che fa la richiesta. Questo campo può essere usato dal server per adattare la risposta ai bisogni di uno specifico client, ma è più spesso usato per produrre statistiche sui client che usano il server.

Il campo `Accept` ha un valore più immediato, perché informa il server sul formato della risorsa richiesta. Se il client è indifferente al formato della risorsa, il campo `Accept` può specificare `*/*` come formato.

Ci sono molti altri campi di intestazione che possono essere usati in un messaggio HTTP, ma i campi mostrati nell'esempio sono sufficienti per richiedere una risorsa al server.

Oltre ai campi dell'intestazione della richiesta, il client può includere altri dati complementari nella richiesta HTTP che sarà inviata al server. Se questi dati consistono solo in semplici parametri di testo, nel formato `nome=valore`, possono essere aggiunti al percorso del metodo GET. I parametri sono incorporati nel percorso dopo un punto interrogativo e sono separati da caratteri *ampersand* (`&`):

```
GET /cgi-bin/receive.cgi?name=LPI&email=info@lpi.org HTTP/1.1
```

In questo esempio, `/cgi-bin/receive.cgi` è il percorso dello script sul server che elaborerà ed eventualmente utilizzerà i parametri `name` e `email`, ottenuti dal percorso della richiesta. La stringa che corrisponde ai campi, nel formato `name=LPI&email=info@lpi.org`, è chiamata *query string* ed è fornita allo script `receive.cgi` dal server HTTP che riceve la richiesta.

Quando i dati sono costituiti da brevi campi di testo, è più appropriato inviarli nel payload del messaggio. In questo caso si deve usare il metodo HTTP POST affinché il server riceva ed elabori il payload del messaggio, secondo le specifiche indicate nell'intestazione della richiesta. Quando si usa il metodo POST, l'intestazione della richiesta deve fornire la dimensione del carico utile che verrà inviato in seguito e come viene formattato il corpo:

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
Content-Length: 1503
Content-Type: multipart/form-data; boundary=-----
405f7edfd646a37d
```

Il campo `Content-Length` indica la dimensione in byte del carico utile e il campo `Content-Type` indica il suo formato. Il formato `multipart/form-data` è quello più comunemente usato nei moduli HTML tradizionali che usano il metodo POST. In questo formato, ogni campo inserito nel payload della richiesta è separato dal codice indicato dalla parola chiave `boundary`. Il metodo POST dovrebbe essere usato solo quando è appropriato, poiché utilizza una quantità di dati leggermente maggiore rispetto a una richiesta equivalente fatta con il metodo GET. Poiché il metodo GET invia i parametri direttamente nell'intestazione del messaggio della richiesta, lo scambio totale di dati ha una latenza inferiore, perché non sarà necessario uno stadio di connessione aggiuntivo per trasmettere il corpo del messaggio.

L'Intestazione di Risposta

Dopo che il server HTTP ha ricevuto l'intestazione del messaggio di richiesta, il server restituisce un messaggio di risposta al client. Una richiesta di file HTML ha tipicamente un'intestazione di risposta come questa:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 18170
Content-Type: text/html
Date: Mon, 05 Apr 2021 13:44:25 GMT
Etag: "606adcd4-46fa"
Last-Modified: Mon, 05 Apr 2021 09:48:04 GMT
Server: nginx/1.17.10
```

La prima riga fornisce la versione del protocollo HTTP usata nel messaggio di risposta, che deve corrispondere alla versione usata nell'intestazione della richiesta. Poi, sempre nella prima riga, appare il codice di stato della risposta, che indica come il server ha interpretato e generato la risposta per la richiesta.

Il codice di stato è un numero di tre cifre, dove la cifra più a sinistra definisce la classe di risposta. Ci sono cinque classi di codici di stato, numerate da 1 a 5, ognuna delle quali indica un tipo di azione intrapresa dal server:

1xx (Informational)

La richiesta è stata ricevuta, continuando il processo.

2xx (Successful)

La richiesta è stata ricevuta, compresa e accettata con successo.

3xx (Redirection)

È necessario intraprendere ulteriori azioni per completare la richiesta.

4xx (Client Error)

La richiesta contiene una sintassi sbagliata o non può essere soddisfatta.

5xx (Server Error)

Il server non è riuscito a soddisfare una richiesta apparentemente valida.

La seconda e la terza cifra sono usate per indicare ulteriori dettagli. Il codice 200, per esempio, indica che è stato possibile rispondere alla richiesta senza problemi. Come mostrato nell'esempio, può essere fornita anche una breve descrizione testuale dopo il codice di risposta (OK). Alcuni codici specifici sono di particolare interesse per assicurare che il client HTTP possa accedere alla risorsa in situazioni avverse o per aiutare a identificare il motivo del fallimento nel caso di una richiesta non riuscita:

301 Moved Permanently

Alla risorsa di destinazione è stato assegnato un nuovo URL permanente, fornito dal campo di intestazione `Location` nella risposta.

302 Found

La risorsa di destinazione risiede temporaneamente sotto un URL diverso.

401 Unauthorized

La richiesta non è stata applicata perché manca di credenziali di autenticazione valide per la risorsa di destinazione.

403 Forbidden

La risposta `Forbidden` indica che, sebbene la richiesta sia valida, il server è configurato per non fornirla.

404 Not Found

Il server di origine non ha trovato una rappresentazione corrente per la risorsa di destinazione o non è disposto a rivelare che ne esista una.

500 Internal Server Error

Il server ha incontrato una condizione inaspettata che gli ha impedito di soddisfare la richiesta.

502 Bad Gateway

Il server, mentre agiva come gateway o proxy, ha ricevuto una risposta non valida da un server in entrata a cui ha avuto accesso mentre tentava di soddisfare la richiesta.

Pur indicando che non è stato possibile soddisfare la richiesta, i codici di stato 4xx e 5xx indicano che il server HTTP è in funzione ed è in grado di ricevere richieste. I codici 4xx richiedono un'azione da parte del client, perché l'URL o le credenziali sono sbagliate. Al contrario, i codici 5xx indicano qualcosa di sbagliato sul lato server. Quindi, nel contesto delle applicazioni web, queste due classi di codici di stato indicano che la fonte dell'errore si trova nell'applicazione stessa, sia client sia server, *non* nell'infrastruttura sottostante.

Contenuto Statico e Dinamico

I server HTTP utilizzano due meccanismi di base per soddisfare il contenuto richiesto dal client. Il primo meccanismo fornisce *contenuto statico*: cioè, il percorso indicato nel messaggio di richiesta corrisponde a un file sul file system locale del server. Il secondo meccanismo fornisce *contenuto dinamico*: cioè, il server HTTP inoltra la richiesta a un altro programma - probabilmente uno script - per costruire la risposta da diverse fonti, come database e altri file.

Anche se ci sono diversi server HTTP, tutti usano lo stesso protocollo di comunicazione HTTP e adottano più o meno le stesse convenzioni. Un'applicazione che non ha una necessità specifica può essere implementata con qualsiasi server tradizionale, come Apache o NGINX. Entrambi sono in grado di generare contenuti dinamici e fornire contenuti statici, ma ci sono sottili differenze nella configurazione di ciascuno di essi.

La posizione dei file statici da servire, per esempio, è definita in modi diversi in Apache e NGINX. La convenzione è di tenere questi file in una directory specifica per questo scopo, con un nome associato all'host, per esempio `/var/www/learning.lpi.org/`. In Apache, questo percorso è definito dalla direttiva di configurazione `DocumentRoot /var/www/learning.lpi.org`, in una sezione che definisce un host virtuale. In NGINX, la direttiva usata è `root /var/www/learning.lpi.org` in una sezione `server` del file di configurazione.

Qualsiasi server tu scelga, i file in `/var/www/learning.lpi.org/` saranno serviti via HTTP quasi allo stesso modo. Alcuni campi nell'intestazione della risposta e i loro contenuti possono variare tra i due server, ma campi come `Content-Type` devono essere presenti nell'intestazione della risposta e devono essere coerenti su qualsiasi server.

Caching

HTTP è stato progettato per funzionare su qualsiasi tipo di connessione Internet, veloce o lenta. Inoltre, la maggior parte degli scambi HTTP deve attraversare molti nodi di rete a causa dell'architettura distribuita di Internet. Di conseguenza è importante adottare qualche strategia di *caching* dei contenuti per evitare il trasferimento ridondante di contenuti precedentemente scaricati. I trasferimenti HTTP possono funzionare con due tipi base di cache: *condivisa* e *privata*.

Una cache condivisa è usata da più di un singolo client. Per esempio, un grande fornitore di contenuti potrebbe usare cache su server geograficamente distribuiti, in modo che i client ricevano i dati dal server più vicino. Una volta che un client ha fatto una richiesta e la sua risposta è stata memorizzata in una cache condivisa, altri client che fanno la stessa richiesta nella stessa area riceveranno la risposta presente nella cache.

Una cache privata è creata dal client stesso per il suo uso esclusivo. È il tipo di cache che il browser web fa per le immagini, i file CSS, il JavaScript o il documento HTML stesso, in modo che non debbano essere scaricati di nuovo se richiesti in un prossimo futuro.

NOTE

Non tutte le richieste HTTP devono essere messe in cache. Una richiesta che utilizza il metodo POST, per esempio, implica una risposta associata esclusivamente a quella particolare richiesta, quindi il suo contenuto di risposta non dovrebbe essere riutilizzato. Per impostazione predefinita, solo le risposte alle richieste effettuate con il metodo GET vengono messe in cache. Inoltre, solo le risposte con codici di stato conclusivi come 200 (OK), 206 (contenuto parziale), 301 (spostato in modo permanente) e 404 (non trovato) sono adatte alla cache.

Sia la strategia della cache condivisa sia quella della cache privata usano le intestazioni HTTP per controllare come il contenuto scaricato dovrebbe essere messo in cache. Per la cache privata, il client consulta l'intestazione della risposta e verifica se il contenuto nella cache locale corrisponde ancora al contenuto remoto corrente. Se è così, il client rinuncia al trasferimento del payload della risposta e usa la versione locale.

La validità della risorsa in cache può essere valutata in diversi modi. Il server può fornire una data di scadenza nell'intestazione di risposta per la prima richiesta, in modo che il client scarti la risorsa in cache alla fine del termine e la richieda nuovamente per ottenere la versione aggiornata. Tuttavia, il server non è sempre in grado di determinare la data di scadenza di una risorsa, quindi è comune usare il campo intestazione di risposta ETag per identificare la versione della risorsa, per esempio Etag: "606adcd4-46fa".

Per verificare che una risorsa in cache debba essere aggiornata, il client richiede al server solo la sua intestazione di risposta. Se il campo ETag corrisponde a quello della versione memorizzata

localmente, il client riutilizza il contenuto della cache. Altrimenti, il contenuto aggiornato della risorsa viene scaricato dal server.

Le Sessioni HTTP

In un sito web convenzionale o in un'applicazione web, le funzioni che gestiscono il controllo della sessione sono basate sulle intestazioni HTTP. Il server non può assumere, per esempio, che tutte le richieste provenienti dallo stesso indirizzo IP siano dello stesso client. Il metodo più tradizionale che permette al server di associare diverse richieste a un unico client è l'uso di *cookies*, un tag di identificazione che viene dato al client dal server e che viene fornito nell'intestazione HTTP.

I cookie permettono al server di conservare informazioni su un client specifico, anche se la persona che lo esegue non si identifica esplicitamente. Con i cookie è possibile implementare sessioni in cui i login, i carrelli della spesa, le preferenze, ecc. vengono conservati tra diverse richieste fatte allo stesso server che li ha forniti. I cookie vengono utilizzati anche per tracciare la navigazione degli utenti, quindi è importante chiedere il consenso prima di inviarli.

Il server imposta il cookie nell'intestazione della risposta usando il campo `Set-Cookie`. Il valore del campo è una coppia `nome=valore` scelta per rappresentare qualche attributo associato a uno specifico client. Il server può, per esempio, creare un numero di identificazione per un client che richiede una risorsa per la prima volta e passarlo al client nell'intestazione di risposta:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Set-Cookie: client_id=62b5b719-fcbf
```

Se il client permette l'uso di cookie, le nuove richieste a questo stesso server hanno il campo cookie nell'intestazione:

```
GET /en/ HTTP/1.1
Host: learning.lpi.org
Cookie: client_id=62b5b719-fcbf
```

Con questo numero di identificazione, il server può recuperare definizioni specifiche per il client e generare una risposta personalizzata. È anche possibile utilizzare più di un campo `Set-Cookie` per fornire diversi cookie allo stesso client. In questo modo, più di una definizione può essere conservata sul lato client.

I cookie sollevano sia problemi di privacy sia potenziali falle di sicurezza, perché c'è la possibilità che possano essere trasferiti a un altro client, che sarà identificato dal server come il client originale. I

cookie utilizzati per preservare le sessioni possono dare accesso a informazioni sensibili del client originale. Pertanto, è molto importante che i client adottino meccanismi di protezione locale per evitare che i loro cookie siano estratti e riutilizzati senza autorizzazione.

Esercizi Guidati

1. Quale metodo HTTP usa il seguente messaggio di richiesta?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

2. Quando un server HTTP ospita molti siti web, come è in grado di identificare quello a cui è destinata una richiesta?

3. Quale parametro è fornito dalla stringa di query dell'URL
<https://www.google.com/search?q=LPI>

4. Perché la seguente richiesta HTTP non è adatta al caching?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

Esercizi Esplorativi

1. Come potresti usare il browser web per monitorare le richieste e le risposte di una pagina HTML?

2. I server HTTP che forniscono contenuti statici di solito mappano il percorso richiesto su un file nel filesystem del server. Che cosa succede quando il percorso nella richiesta punta a una directory?

3. Il contenuto dei file inviati tramite HTTPS è protetto da crittografia, quindi non può essere letto dai computer tra il client e il server. Nonostante ciò, questi computer nel mezzo possono identificare quale risorsa il client ha richiesto al server?

Sommario

Questa lezione tratta le basi di HTTP, il principale protocollo utilizzato dalle applicazioni client per richiedere risorse ai server web. La lezione passa attraverso i seguenti concetti:

- Messaggi di richiesta, campi di intestazione e metodi.
- Codici di stato delle risposte.
- Come i server HTTP generano le risposte.
- Caratteristiche HTTP utili per il caching e la gestione delle sessioni.

Risposte agli Esercizi Guidati

- Quale metodo HTTP usa il seguente messaggio di richiesta?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

Il metodo POST.

- Quando un server HTTP ospita molti siti web, come è in grado di identificare quello a cui è destinata una richiesta?

Il campo Host nell'intestazione della richiesta fornisce il sito web mirato.

- Quale parametro è fornito dalla stringa di query dell'URL <https://www.google.com/search?q=LPI>?

Il parametro chiamato q con un valore di LPI.

- Perché la seguente richiesta HTTP non è adatta al caching?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

Poiché le richieste fatte con il metodo POST implicano un'operazione di scrittura sul server, non dovrebbero essere messe in cache.

Risposte agli Esercizi Esplorativi

1. Come potresti usare il browser web per monitorare le richieste e le risposte di una pagina HTML?

Tutti i browser popolari offrono *strumenti di sviluppo* che, tra le altre cose, possono mostrare tutte le transazioni di rete che sono state effettuate dalla pagina corrente.

2. I server HTTP che forniscono contenuti statici di solito mappano il percorso richiesto su un file nel filesystem del server. Che cosa succede quando il percorso nella richiesta punta a una directory?

Dipende da come è configurato il server. Per default, la maggior parte dei server HTTP cerca un file chiamato `index.html` (o un altro nome predefinito) in quella stessa directory e lo invia come risposta. Se il file non c'è, il server emette una risposta `404 Not Found`.

3. Il contenuto dei file inviati tramite HTTPS è protetto da crittografia, quindi non può essere letto dai computer tra il client e il server. Nonostante ciò, questi computer nel mezzo possono identificare quale risorsa il client ha richiesto al server?

No, perché le stesse intestazioni HTTP di richiesta e risposta sono criptate da TLS.



Argomento 032: Marcatura di un Documento HTML



032.1 Anatomia di un Documento HTML

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 032.1

Peso

2

Arese di Conoscenza Chiave

- Creare un semplice documento HTML
- Comprendere il ruolo dell'HTML
- Comprendere la struttura dell'HTML
- Comprendere la sintassi HTML (tag, attributi, commenti)
- Comprendere l'intestazione dell'HTML
- Comprendere i meta tag
- Comprendere la codifica dei caratteri

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- `<!DOCTYPE html>`
- `<html>`
- `<head>`
- `<body>`
- `<meta>`, incluso `charset (UTF-8)`, `name` e `content` degli attributi



032.1 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	032 Marcatura di un Documento HTML
Obiettivo:	032.1 Anatomia di un Documento HTML
Lezione:	1 di 1

Introduzione

HTML (*HyperText Markup Language*) è un linguaggio di marcatura (*markup*) che indica ai browser web come strutturare e visualizzare le pagine web. La versione corrente è la 5.0, che è stata rilasciata nel 2012. La sintassi HTML è definita dal *World Wide Web Consortium* (W3C).

L'HTML è un'abilità fondamentale nello sviluppo web, poiché definisce la struttura e buona parte dell'aspetto di un sito web. Se vuoi una carriera nello sviluppo web, l'HTML è sicuramente un buon punto di partenza.

Anatomia di un Documento HTML

Una pagina HTML di base ha la seguente struttura:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My HTML Page</title>
    <!-- This is the Document Header -->
  </head>

  <body>
    <!-- This is the Document Body -->
  </body>
</html>
```

Ora, analizziamola in dettaglio.

Tag HTML

L'HTML usa *elementi* e *tag* per descrivere e formattare il contenuto. I tag consistono in parentesi angolari intorno al nome di un tag, per esempio `<title>`. Il nome del tag non è sensibile alle maiuscole, sebbene il World Wide Web Consortium (W3C) raccomandi di usare lettere minuscole nelle versioni correnti dell'HTML. Questi tag HTML sono usati per costruire elementi. Il tag `<title>` è un esempio per un *tag di apertura* di un elemento HTML che definisce il titolo di un documento. Tuttavia, un elemento ha due ulteriori componenti. Un elemento completo `<title>` che assomiglia a questo:

```
<title>My HTML Page</title>
```

Nel caso precedente, `My HTML Page` serve come elemento *contenuto*, mentre `</title>` serve come *tag di chiusura* che dichiara che questo elemento è completo.

NOTE

Non tutti gli elementi HTML hanno bisogno di essere chiusi; in questi casi, si parla di elementi vuoti, elementi auto-chiudenti, o elementi nulli.

Ecco gli altri elementi HTML dell'esempio precedente:

`<html>`

Racchiude l'intero documento HTML. Contiene tutti i tag che compongono la pagina. Indica anche che il contenuto di questo file è in linguaggio HTML. Il suo tag di chiusura corrispondente è `</html>`.

<head>

Un contenitore per tutte le informazioni meta riguardanti la pagina. Il tag di chiusura corrispondente di questo elemento è </head>.

<body>

Un contenitore per il contenuto della pagina e la sua rappresentazione strutturale. Il suo tag di chiusura corrispondente è </body>.

I tag <html>, <head>, <body> e <title> sono i cosiddetti tag *scheletro*, che forniscono la struttura di base di un documento HTML. In particolare, dicono al browser web che sta leggendo una pagina HTML.

NOTE

Di questi elementi, l'unico che è richiesto per la validazione di un documento HTML è il tag <title>.

Come puoi vedere, ogni pagina HTML è un documento ben strutturato e potrebbe anche essere definito come un albero, dove l'elemento <html> rappresenta la radice del documento e gli elementi <head> e <body> sono i primi rami. L'esempio mostra che è possibile annidare elementi: Per esempio, l'elemento <title> è annidato dentro l'elemento <head>, che a sua volta è annidato dentro l'elemento <html>.

Per essere sicuri che il codice HTML sia leggibile e mantenibile, assicurati che tutti gli elementi HTML siano chiusi correttamente e in ordine. I browser possono ancora visualizzare il tuo sito web come previsto, ma l'annidamento non corretto degli elementi e dei loro tag è una pratica soggetta a errori.

Infine, una menzione speciale va alla dichiarazione *doctype* all'inizio della struttura del documento di esempio. <!DOCTYPE> *non* è un tag HTML, ma un'istruzione per il browser web che specifica la versione HTML usata nel documento. Nella struttura base del documento HTML mostrato prima, è stato usato <!DOCTYPE html>, specificando che HTML5 è usato in questo documento.

Commenti nell'HTML

Quando si crea una pagina HTML, è buona pratica inserire dei commenti nel codice per migliorarne la leggibilità e descrivere lo scopo dei blocchi di codice più grandi. Un commento viene inserito tra i tag <!-- e -->, come mostrato nel seguente esempio:

```
<!-- This is a comment. -->  
  
<!--  
    This is a  
    multiline  
    comment.  
-->
```

L'esempio dimostra che i commenti HTML possono essere inseriti in una singola riga, ma possono anche estendersi su più righe. In ogni caso, il risultato è che il testo tra `<!--` e `-->` viene ignorato dal browser web e quindi non viene visualizzato nella pagina HTML. Sulla base di queste considerazioni, puoi dedurre che la pagina HTML di base mostrata nella sezione precedente non mostra alcun testo, perché le linee `<!-- This is the Document Header -->` e `<!-- This is the Document Body -->` sono solo due commenti.

WARNING

I commenti non possono essere annidati.

Attributi HTML

I tag HTML possono includere uno o più *attributi* per specificare i dettagli dell'elemento HTML. Un semplice tag con due attributi ha la seguente forma:

```
<tag attribute-a="value-a" attribute-b="value-b">
```

Gli attributi devono sempre essere impostati sul tag di apertura.

Un attributo consiste in un nome, che indica la proprietà che dovrebbe essere impostata, un segno di uguale e il valore desiderato tra virgolette. Sia le virgolette singole sia quelle doppie sono accettabili, ma si raccomanda di usare le virgolette singole o doppie in modo coerente in tutto il progetto. È importante non mischiare apici singoli e doppi per un singolo valore di attributo, poiché il browser web non riconoscerà gli apici misti come un'unità.

NOTE

Puoi includere un tipo di virgolette all'interno dell'altro tipo senza problemi. Per esempio, se hai bisogno di usare `'` in un valore di attributo, puoi includere quel valore dentro `"`. Tuttavia, se vuoi usare lo stesso tipo di virgolette all'interno del valore che stai usando per includere il valore, devi usare `"` per `"` e `'` per `'`.

Gli attributi possono essere categorizzati in *attributi fondamentali* e *attributi specifici* come spiegato

nelle sezioni seguenti.

Attributi Fondamentali

Gli attributi fondamentali sono quelli che possono essere usati su qualsiasi elemento HTML. Essi includono:

title

Describe il contenuto dell'elemento. Il suo valore è spesso visualizzato come un *tooltip* che viene mostrato quando l'utente sposta il suo cursore sull'elemento.

id

Associa un identificatore unico a un elemento. Questo identificatore deve essere unico all'interno del documento, e il documento non verrà validato quando più elementi condividono lo stesso **id**.

style

Assegna proprietà grafiche (stili CSS) all'elemento.

class

Specificata una o più classi per l'elemento in una lista separata da spazi di nomi di classi. Queste classi possono essere referenziate nei fogli di stile CSS.

lang

Specifica la lingua del contenuto dell'elemento usando i codici di lingua a due caratteri dello standard ISO-639.

NOTE

Lo sviluppatore può memorizzare informazioni personalizzate su un elemento definendo un cosiddetto attributo `data-`, che viene indicato facendo precedere il nome desiderato da `data-` come in `data-additionalinfo`. È possibile assegnare a questo attributo un valore proprio come qualsiasi altro attributo.

Attributi Specifici

Altri attributi sono specifici per ogni elemento HTML. Per esempio, l'attributo `src` di un elemento HTML `` specifica l'URL di un'immagine. Ci sono molti altri attributi specifici, che saranno trattati nelle prossime lezioni.

Intestazione del Documento

L'intestazione del documento definisce le *meta* informazioni riguardanti la pagina ed è descritta

dall'elemento `<head>`. Per impostazione predefinita, le informazioni all'interno dell'intestazione del documento non vengono visualizzate dal browser web. Mentre è possibile usare l'elemento `<head>` per contenere elementi HTML che potrebbero essere visualizzati nella pagina, farlo non è raccomandato.

Il Titolo

Il titolo del documento è specificato usando l'elemento `<title>`. Il titolo definito tra i tag appare nella barra del titolo del web browser ed è il nome suggerito per il segnalibro. Viene anche visualizzato nei risultati dei motori di ricerca come titolo della pagina.

Un esempio di questo elemento è il seguente:

```
<title>My test page</title>
```

Il tag `<title>` è richiesto in tutti i documenti HTML e dovrebbe apparire solo una volta in ogni documento.

NOTE

Non confondete il titolo del documento con l'intestazione della pagina, che si trova nel corpo.

Metadati

L'elemento `<meta>` è usato per specificare *meta* informazioni per descrivere ulteriormente il contenuto di un documento HTML. È un cosiddetto elemento auto-chiudente, il che significa che non ha un tag di chiusura. A parte gli attributi fondamentali che sono validi per ogni elemento HTML, l'elemento `<meta>` usa anche i seguenti attributi:

name

Definisce quali metadati saranno descritti in questo elemento. Può essere impostato su qualsiasi valore definito dall'utente, ma i valori comunemente usati sono `author`, `description` e `keywords`.

http-equiv

Fornisce un'intestazione HTTP per il valore dell'attributo `content`. Un valore comune è `refresh`, che sarà spiegato più avanti. Se questo attributo è impostato, l'attributo `name` non dovrebbe esserlo.

content

Fornisce il valore associato all'attributo `name` o `http-equiv`.

charset

Specifica la codifica dei caratteri per il documento HTML, per esempio `utf-8` per impostarlo su Unicode Transformation Format – 8-bit.

Aggiungere un Autore, una Descrizione e Parole Chiave

Usando il tag `<meta>`, puoi specificare informazioni aggiuntive sull'autore della pagina HTML e descrivere il contenuto della pagina in questo modo:

```
<meta name="author" content="Name Surname">
<meta name="description" content="A short summary of the page content.">
```

Cerca di includere nella descrizione una serie di parole chiave legate al contenuto della pagina. Questa descrizione è spesso la prima cosa che un utente vede quando naviga utilizzando un motore di ricerca.

Se vuoi quindi fornire ulteriori parole chiave ai motori di ricerca relative alla pagina web, potete aggiungere questo elemento:

```
<meta name="keywords" content="keyword1, keyword2, keyword3, keyword4, keyword5">
```

NOTE

In passato, gli spammer inserivano centinaia di parole chiave e descrizioni estranee al contenuto effettivo della pagina, in modo che questa apparisse anche nelle ricerche non correlate ai termini ricercati dalle persone. Oggi i tag `<meta>` sono relegati in una posizione di secondaria importanza e vengono utilizzati solo per consolidare gli argomenti trattati nella pagina web, in modo che non sia più possibile ingannare i nuovi e più sofisticati algoritmi dei motori di ricerca.

Reindirizzare una Pagina HTML e Definire un Intervallo di Tempo in cui il Documento si Aggiorerà da Solo

Usando il tag `<meta>`, puoi aggiornare automaticamente una pagina HTML dopo un certo periodo (per esempio dopo 30 secondi) in questo modo:

```
<meta http-equiv="refresh" content="30">
```

In alternativa, puoi reindirizzare una pagina web a un'altra pagina dopo lo stesso tempo con il seguente codice:

```
<meta http-equiv="refresh" content="30; url=http://www.lpi.org">
```

Nel precedente esempio, l'utente viene reindirizzato dalla pagina corrente a <http://www.lpi.org> dopo 30 secondi. I valori possono essere variati a piacere. Per esempio, se si specifica content="0; url=http://www.lpi.org", la pagina viene reindirizzata immediatamente.

Specificare la Codifica dei Caratteri

L'attributo charset specifica la codifica dei caratteri per il documento HTML. Un esempio comune è:

```
<meta charset="utf-8">
```

Questo elemento specifica che la codifica dei caratteri del documento è `utf-8`, che è un set di caratteri universale che include praticamente qualsiasi carattere di qualsiasi lingua umana. Pertanto, usandolo, si eviteranno problemi nella visualizzazione di alcuni caratteri che si possono avere usando altri set di caratteri come ISO-8859-1 (l'alfabeto latino).

Altri Esempi Utili

Altre due utili applicazioni del tag `<meta>` sono:

- Impostare i cookie per tenere traccia del visitatore di un sito.
- Prendere il controllo del *viewport* (l'area visibile di una pagina web all'interno della finestra di un browser web), che dipende dalle dimensioni dello schermo del dispositivo dell'utente (per esempio, un telefono cellulare o un computer).

Tuttavia, questi due esempi sono al di là dello scopo dell'esame e il loro studio è lasciato al lettore curioso di esplorare ulteriori funzionalità.

Esercizi Guidati

1. Per ognuno dei seguenti tag, indica il tag di chiusura corrispondente:

<body>	
<head>	
<html>	
<meta>	
<title>	

2. Qual è la differenza tra un tag e un elemento? Usa questa voce come riferimento:

```
<title>HTML Page Title</title>
```

3. Quali sono i tag tra cui inserire un commento?

4. Spiega cos'è un attributo e fornisci alcuni esempi per il tag <meta>.

Esercizi Esplorativi

1. Crea un semplice documento HTML versione 5 con il titolo My first HTML document e un solo paragrafo nel corpo, contenente il testo Hello World. Usa il tag paragrafo `<p>` nel corpo.

2. Aggiungi l'autore (Kevin Author) e la descrizione (This is my first HTML page.) del documento HTML.

3. Aggiungi le seguenti parole chiave relative al documento HTML: HTML, Example, Test, e Metadata.

4. Aggiungi l'elemento `<meta charset="ISO-8859-1">` all'intestazione del documento e cambia il testo Hello World con quello in giapponese (世界). Che cosa succede? Come puoi risolvere il problema?

5. Dopo aver cambiato il testo del paragrafo in Hello World, reindirizza la pagina HTML a <https://www.google.com> dopo 30 secondi e aggiungi un commento che spieghi questo nell'intestazione del documento.

Sommario

In questa lezione hai imparato:

- Il ruolo dell'HTML
- Lo struttura dell'HTML
- La sintassi dell'HTML (tag, attributi, commenti)
- L'intestazione dell'HTML
- I meta tag
- Come creare un semplice documento HTML

I seguenti termini sono stati discussi in questa lezione:

<!DOCTYPE html>

Il tag di dichiarazione.

<html>

Il contenitore di tutti i tag che compongono la pagina HTML.

<head>

Il contenitore di tutti gli elementi di intestazione.

<body>

Il contenitore di tutti gli elementi del corpo.

<meta>

Il tag per i metadati, usato per specificare informazioni aggiuntive per la pagina HTML (come autore, descrizione e codifica dei caratteri).

Risposte agli Esercizi Guidati

1. Per ognuno dei seguenti tag, indica il tag di chiusura corrispondente:

<body>	</body>
<head>	</head>
<html>	</html>
<meta>	None
<title>	</title>

2. Qual è la differenza tra un tag e un elemento? Usa questa voce come riferimento:

```
<title>HTML Page Title</title>
```

Un elemento HTML consiste in un tag iniziale, un tag di chiusura e tutto ciò che sta nel mezzo. Un tag HTML è usato per marcare l'inizio o la fine di un elemento. Perciò, `<title>HTML Page Title</title>` è un elemento HTML, mentre `<title>` e `</title>` sono rispettivamente il tag iniziale e quello di chiusura.

3. Quali sono i tag tra cui inserire un commento?

Un commento è inserito tra i tag `<!--` e `-->` e può essere messo su una singola linea o può abbracciare più linee.

4. Spiega cos'è un attributo e fornisci alcuni esempi per il tag `<meta>`.

Un attributo è usato per specificare più precisamente un elemento HTML. Per esempio, il tag `<meta>` usa la coppia di attributi `name` e `content` per aggiungere l'autore e la descrizione di una pagina HTML. Invece, usando l'attributo `charset` puoi specificare la codifica dei caratteri per il documento HTML.

Risposte agli Esercizi Esplorativi

1. Crea un semplice documento HTML versione 5 con il titolo My first HTML document e un solo paragrafo nel corpo, contenente il testo Hello World. Usa il tag paragrafo `<p>` nel corpo.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

2. Aggiungi l'autore (Kevin Author) e la descrizione (This is my first HTML page.) del documento HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

3. Aggiungi le seguenti parole chiave relative al documento HTML: HTML, Example, Test, e Metadata.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
    <meta name="keywords" content="HTML, Example, Test, Metadata">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

4. Aggiungi l'elemento `<meta charset="ISO-8859-1">` all'intestazione del documento e cambia il testo `Hello World` con quello in giapponese (世界). Che cosa succede? Come puoi risolvere il problema?

Se l'esempio viene eseguito come descritto, il testo giapponese non viene visualizzato correttamente. Questo perché ISO-8859-1 rappresenta la codifica dei caratteri per l'alfabeto latino. Per visualizzare il testo, è necessario cambiare la codifica dei caratteri, usando per esempio UTF-8 (`<meta charset="utf-8">`).

5. Dopo aver cambiato il testo del paragrafo in `Hello World`, reindirizza la pagina HTML a `https://www.google.com` dopo 30 secondi e aggiungi un commento che spieghi questo nell'intestazione del documento.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
    <meta name="keywords" content="HTML, Example, Test, Metadata">
    <meta charset="utf-8">
    <!-- The page is redirected to Google after 30 seconds -->
    <meta http-equiv="refresh" content="30; url=https://www.google.com">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```



032.2 Semantica HTML e Gerarchia del Documento

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 032.2

Peso

2

Arese di Conoscenza Chiave

- Creare marature per i contenuti in un documento HTML
- Comprendere la struttura gerarchica del testo HTML
- Differenziare tra elementi HTML a blocchi e in linea
- Comprendere importanti elementi strutturali HTML semantici

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- <h1>, <h2>, <h3>, <h4>, <h5>, <h6>
- <p>
- , ,
- <dl>, <dt>, <dd>
- <pre>
- <blockquote>
- , , <code>
- , <i>, <u>
-

- <div>
- <main>, <header>, <nav>, <section>, <footer>



032.2 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	032 Marcatura di un Documento HTML
Obiettivo:	032.2 Semantic HTML e Gerarchia del Documento
Lezione:	1 di 1

Introduzione

Nella lezione precedente abbiamo imparato che l'HTML è un linguaggio di *markup* che può descrivere semanticamente il contenuto di un sito web. Un documento HTML contiene un cosiddetto *skeleton* che consiste degli elementi HTML: `<html>`, `<head>`, e `<body>`. Mentre l'elemento `<head>` descrive un blocco di informazioni *meta* per il documento HTML che sarà invisibile al visitatore del sito web, l'elemento `<body>` può contenere molti altri elementi per definire la struttura e il contenuto del documento HTML.

In questa lezione, ci occuperemo di formattazione del testo, elementi HTML semantici fondamentali e loro scopo, e struttura di un documento HTML.

NOTE

Tutti gli esempi di codice successivi si trovano all'interno dell'elemento `<body>` di un documento HTML che contiene la struttura (*skeleton*) completa. Per una migliore leggibilità, *non* mostreremo l'HTML completo in ogni esempio di questa lezione.

Il Testo

Nell'HTML, nessun blocco di testo dovrebbe essere al di fuori di un elemento. Anche un breve paragrafo dovrebbe essere compreso nei tag HTML `<p>`, che è il nome breve di *paragrafo*.

```
<p>Short text element spanning only one line.</p>
<p>A text element containing much longer text that may span across multiple lines,
depending on the size of the web browser window.</p>
```

Aperto in un browser web, questo codice HTML produce il risultato mostrato in [Figure 1](#).

Short text element spanning only one line

A text element containing much longer text that may span across multiple lines depending on the size of the web browser window.

Figure 1. Rappresentazione del browser web del codice HTML che mostra due paragrafi di testo. Il primo paragrafo è molto breve. Il secondo paragrafo è un po' più lungo tanto da creare una seconda riga.

Per default i browser web aggiungono una spaziatura prima e dopo gli elementi `<p>` per migliorare la leggibilità. Per questa ragione, `<p>` è chiamato un *elemento di blocco*.

Le Intestazioni

L'HTML definisce sei livelli di intestazioni per descrivere e strutturare il contenuto di un documento HTML. Queste intestazioni sono contrassegnate dai tag HTML `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` e `<h6>`.

```
<h1>Livello di intestazione 1 per identificare univocamente la pagina</h1>
<h2>Livello di intestazione 2</h2>
<h3>Livello di intestazione 3</h3>
<h4>Livello di intestazione 4</h4>
<h5>Livello di intestazione 5</h5>
<h6>Livello di intestazione 6</h6>
```

Un browser web rende questo codice HTML come mostrato in [Figure 2](#).

Headline level 1 to uniquely identify the page

Headline level 2

Headline level 3

Headline level 4

Headline level 5

Headline level 6

Figure 2. Rappresentazione del browser web del codice HTML che mostra diversi livelli di intestazioni in un documento HTML. La gerarchia delle intestazioni è indicata attraverso la dimensione del testo.

Se hai familiarità con i word processor come LibreOffice o Microsoft Word, potresti notare alcune somiglianze nel modo in cui un documento HTML usa diversi livelli di intestazione e come vengono resi nel browser web. Per impostazione predefinita, l'HTML usa le dimensioni per indicare la gerarchia e l'importanza dei titoli e aggiunge spazio prima e dopo ogni titolo per separarlo visivamente dal contenuto.

Un'intestazione che usa l'elemento `<h1>` è in cima alla gerarchia e quindi è considerata l'intestazione più importante che identifica il contenuto della pagina. È paragonabile all'elemento `<title>` discusso nella lezione precedente, ma *all'interno* del contenuto del documento HTML. Gli elementi di intestazione successivi possono essere usati per strutturare ulteriormente il contenuto. Assicurati di non saltare i livelli di intestazione in mezzo. La gerarchia del documento dovrebbe iniziare con `<h1>`, continuare con `<h2>`, poi `<h3>` e così via. Non c'è bisogno di usare ogni elemento di intestazione fino a `<h6>` se il tuo contenuto non lo richiede.

NOTE

Le intestazioni sono strumenti importanti per strutturare un documento HTML, sia semanticamente sia visivamente. Comunque, i titoli non dovrebbero mai essere usati per aumentare la dimensione di un testo strutturalmente non importante. Per lo stesso principio, non si dovrebbe rendere un breve paragrafo in grassetto o in corsivo per farlo sembrare un titolo; usate i tag di intestazione per contrassegnare i titoli.

Cominciamo a creare il documento HTML della lista della spesa definendo il suo contorno. Creeremo un elemento `<h1>` per contenere il titolo della pagina, in questo caso *Garden Party*, seguito da brevi

informazioni avvolte in un elemento `<p>`. Inoltre, usiamo due elementi `<h2>` per introdurre le due sezioni di contenuto `Agenda` e `Please bring`.

```
<h1>Garden Party</h1>
<p>Invitation to John's garden party on Saturday next week.</p>
<h2>Agenda</h2>
<h2>Please bring</h2>
```

Aperto in un browser web, questo codice HTML produce il risultato mostrato in [Figure 3](#).

Garden Party

Invitation to John's garden party on Saturday next week.

Agenda

Please bring

Figure 3. rappresentazione per browser web del codice HTML che mostra un semplice documento di esempio che descrive un invito a una festa in giardino, con due intestazioni per l'ordine del giorno e la lista delle cose da portare.

Interruzioni di Linea

A volte può essere necessario generare un'*interruzione di linea* senza inserire un altro elemento `<p>` o qualsiasi altro elemento di blocco simile. In questi casi, puoi usare l'elemento di chiusura automatica `
`. Nota che dovrebbe essere usato solo per inserire interruzioni di riga che appartengono al contenuto, come nel caso di poesie, testi di canzoni o indirizzi. Se il contenuto è separato dal significato, è meglio usare invece un elemento `<p>`.

Per esempio, potremmo dividere il testo del paragrafo informativo del nostro esempio precedente come segue:

```
<p>
  Invitation to John's garden party.<br>
  Saturday, next week.
</p>
```

In un browser web, questo codice HTML produce il risultato mostrato in [Figure 4](#).

Invitation to John's garden party.
Saturday, next week.

Figure 4. Rappresentazione del browser web del codice HTML che mostra un semplice documento di esempio con un'interruzione di linea forzata.

Linee Orizzontali

L'elemento `<hr>` definisce una linea orizzontale, chiamata anche *regolo orizzontale*. Per impostazione predefinita, abbraccia l'intera larghezza del suo elemento padre. L'elemento `<hr>` può aiutarti a definire un cambiamento tematico nel contenuto o a separare le sezioni del documento. L'elemento è *auto-chiudente* e quindi *non* ha un tag di chiusura.

Per il nostro esempio, potremmo separare i due titoli:

```
<h1>Garden Party</h1>
<p>Invitation to John's garden party on Saturday next week.</p>
<h2>Agenda</h2>
<hr>
<h2>Please bring</h2>
```

[Figure 5](#) mostra il risultato di questo codice.

Garden Party

Invitation to John's garden party on Saturday next week.

Agenda

Please bring

Figure 5. Rappresentazione del browser web di un semplice documento di esempio che descrive una lista della spesa con due sezioni separate da una linea orizzontale.

Le Liste in HTML

In HTML, si possono definire tre tipi di liste:

Liste ordinate

dove l'ordine degli elementi della lista è importante

Liste non ordinate

dove l'ordine degli elementi della lista non è particolarmente importante

Elenchi di descrizione

per descrivere più da vicino alcuni termini

Ognuno dei tipi contiene un numero qualsiasi di *liste*. Descriveremo ogni tipo di lista.

Liste Ordinate

Una *lista ordinata* in HTML, denotata usando l'elemento HTML ``, è una collezione di un qualsiasi numero di *elementi della lista*. Ciò che rende speciale questo elemento è che l'ordine dei suoi elementi di lista è rilevante. Per enfatizzare questo, i browser web visualizzano i numeri prima degli elementi dell'elenco figlio per default.

NOTE

Gli elementi `` sono gli unici elementi figli validi all'interno di un elemento ``.

Per il nostro esempio, possiamo compilare l'agenda della festa in giardino usando un elemento `` con il seguente codice:

```
<h2>Agenda</h2>
<ol>
  <li>Welcome</li>
  <li>Barbecue</li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

In un browser web, questo codice HTML produce il risultato mostrato in [Figure 6](#).

Agenda

1. Welcome
2. Barbecue
3. Dessert
4. Fireworks

Figure 6. Rappresentazione per browser web di un semplice documento di esempio contenente un'intestazione di secondo livello seguita da una lista ordinata con quattro voci che descrivono l'agenda di una festa in giardino.

Opzioni

Come puoi vedere in questo esempio, le voci della lista sono numerate con numeri decimali che iniziano da 1 per impostazione predefinita. Tuttavia, puoi cambiare questo comportamento specificando l'attributo `type` del tag ``. I valori validi per questo attributo sono 1 per i numeri decimali, A per le lettere maiuscole, a per le lettere minuscole, I per i numeri romani maiuscoli, e i per i numeri romani minuscoli.

Se vuoi, puoi anche definire il valore iniziale usando l'attributo `start` del tag ``. L'attributo `start` prende sempre un valore numerico decimale, anche se l'attributo `type` imposta un diverso tipo di numerazione.

Per esempio, potremmo modificare la lista ordinata dell'esempio precedente in modo che le voci della lista siano precedute da lettere maiuscole, iniziando con la lettera C, come mostrato nell'esempio seguente:

```
<h2>Agenda</h2>
<ol type="A" start="3">
  <li>Welcome</li>
  <li>Barbecue</li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

All'interno di un browser web, questo codice HTML è reso come Figure 7.

Agenda

- C. Welcome
- D. Barbecue
- E. Dessert
- F. Fireworks

Figure 7. Il browser web mostra un semplice documento di esempio contenente un'intestazione di secondo livello seguita da una lista ordinata con elementi che sono preceduti da lettere maiuscole che iniziano con la lettera C.

L'ordine delle voci della lista può anche essere invertito usando l'attributo `reversed` senza un valore.

NOTE

In una lista ordinata, puoi anche impostare il valore iniziale di uno specifico elemento della lista usando l'attributo `value` del tag ``. Gli elementi della lista che seguono aumenteranno a partire da quel numero. L'attributo `value` prende sempre un valore numerico decimale.

Liste non Ordinate

Una *lista non ordinata* contiene una serie di elementi della lista che, a differenza di quelli di una lista ordinata, non hanno un ordine o una sequenza speciale. L'elemento HTML per questa lista è ``. Ancora una volta, `` è l'elemento HTML per marcare gli elementi della lista.

NOTE

Gli elementi `` sono gli unici elementi figli validi all'interno di un elemento ``.

Per il nostro sito web di esempio, possiamo usare la lista non ordinata per elencare gli oggetti che gli ospiti devono portare alla festa. Possiamo ottenerlo con il seguente codice HTML:

```
<h2>Please bring</h2>
<ul>
  <li>Salad</li>
  <li>Drinks</li>
  <li>Bread</li>
  <li>Snacks</li>
  <li>Desserts</li>
</ul>
```

All'interno di un browser web, questo codice HTML produce la visualizzazione mostrata in [Figure 8](#).

Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

Figure 8. Visualizzazione per browser web di un semplice documento contenente un'intestazione di secondo livello seguita da una lista non ordinata con voci di lista riguardanti gli alimenti che gli ospiti sono invitati a portare alla festa in giardino.

Per impostazione predefinita, ogni elemento della lista è rappresentato da un pallino. Puoi cambiare il suo aspetto usando i CSS, che saranno discussi nelle lezioni successive.

Liste Annidate

Le liste possono essere annidate dentro altre liste, come le liste ordinate dentro liste non ordinate e viceversa. Per ottenere questo, la lista annidata deve essere parte di un elemento di lista ``, perché `` è l'unico elemento figlio valido di liste non ordinate e ordinate. Quando si annida, fai attenzione a non sovrapporre i tag HTML.

Per il nostro esempio, potremmo aggiungere alcune informazioni dell'agenda che abbiamo creato prima, come mostrato nel seguente esempio:

```
<h2>Agenda</h2>
<ol type="A" start="3">
  <li>Welcome</li>
  <li>
    Barbecue
    <ul>
      <li>Vegetables</li>
      <li>Meat</li>
      <li>Burgers, including vegetarian options</li>
    </ul>
  </li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

Un browser web visualizza il codice come mostrato in [Figure 9](#).

Agenda

- C. Welcome
- D. Barbecue
 - Vegetables
 - Meat
 - Burgers, including vegetarian options
- E. Dessert
- F. Fireworks

Figure 9. Rappresentazione per il browser web del codice HTML che mostra una lista non ordinata annidata dentro una lista ordinata, per rappresentare l'ordine del giorno di una festa in giardino.

Si potrebbe andare anche oltre e annidare più livelli in profondità. Teoricamente, non c'è limite a quante liste puoi annidare. Quando fai questo, però, considera la leggibilità per i tuoi visitatori.

Liste di Descrizione

Una *lista di descrizione* è definita usando l'elemento `<dl>` e rappresenta un dizionario di *chiavi* e *valori*. La chiave è un termine o un nome che vuoi descrivere, e il valore è la descrizione. Le liste di descrizione possono variare da semplici coppie chiave-valore a definizioni estese.

Una chiave (o *termine*) è definita usando l'elemento `<dt>`, mentre la sua descrizione è definita usando l'elemento `<dd>`.

Un esempio per una tale lista di descrizione potrebbe essere una lista di frutti esotici che ne spiega le caratteristiche.

```

<h3>Exotic Fruits</h3>
<dl>
  <dt>Banana</dt>
  <dd>
    A long, curved fruit that is yellow-skinned when ripe. The fruit's skin
    may also have a soft green color when underripe and get brown spots when
    overripe.
  </dd>

  <dt>Kiwi</dt>
  <dd>
    A small, oval fruit with green flesh, black seeds, and a brown, hairy
    skin.
  </dd>

  <dt>Mango</dt>
  <dd>
    A fruit larger than a fist, with a green skin, orange flesh, and one big
    seed. The skin may have spots ranging from green to yellow or red.
  </dd>
</dl>

```

In un browser web, questo produce il risultato mostrato in [Figure 10](#).

Exotic Fruits

Banana

A long, curved fruit that is yellow-skinned when ripe. The fruit's skin may also have a soft green color when underripe and get brown spots when overripe.

Kiwi

A small, oval fruit with green flesh, black seeds and a brown, hairy skin.

Mango

A fruit larger than a fist, with a green skin, orange flesh, and one big seed. The skin may have spots ranging from green to yellow or red.

Figure 10. Un esempio di una lista descrittiva HTML che utilizza frutti esotici. La lista descrive l'aspetto di tre diversi frutti esotici.

NOTE

A differenza delle liste ordinate e non ordinate, in una lista di descrizione, qualsiasi elemento HTML è valido come figlio diretto. Questo ti permette di raggruppare elementi e stilizzarli altrove usando i CSS.

Formattazione del Testo *Inline*

In HTML, si possono usare elementi di formattazione per cambiare l'aspetto del testo. Questi elementi possono essere categorizzati come *elementi di presentazione* o *elementi di frase*.

Elementi di Presentazione

Gli elementi di presentazione di base cambiano il carattere o l'aspetto del testo; questi sono ``, `<i>`, `<u>` e `<tt>`. Questi elementi erano originariamente definiti prima che i CSS permettessero di rendere il testo in grassetto, corsivo, ecc. Ora ci sono di solito modi migliori per alterare l'aspetto del testo, ma è possibile ancora vederli utilizzati.

Testo Grassetto

Per rendere il testo in grassetto, puoi inserirlo all'interno dell'elemento `` come illustrato nel seguente esempio. Il risultato appare in [Figure 11](#).

This `word` is bold.

This **word** is bold.

Figure 11. Il tag `` è usato per rendere il testo in grassetto.

Secondo la specifica HTML5, l'elemento `` dovrebbe essere usato solo quando non ci sono tag più appropriati. L'elemento che produce lo stesso risultato visivo, ma aggiunge importanza semantica al testo marcato, è ``.

Testo Corsivo

Per rendere il testo in corsivo, puoi inserirlo all'interno dell'elemento `<i>` come illustrato nell'esempio seguente. Il risultato appare in [Figure 12](#).

This `<i>word</i>` is in italics.

This *word* is in italics.

Figure 12. Il tag `<i>` è usato per rendere il testo in corsivo.

Secondo la specifica HTML 5, l'elemento `<i>` dovrebbe essere usato solo quando non ci sono tag più appropriati.

Testo Sottolineato

Per sottolineare il testo, puoi inserirlo all'interno dell'elemento `<u>` come illustrato nel seguente esempio. Il risultato appare in [Figure 13](#).

This `<u>word</u>` is underlined.

This word is underlined.

Figure 13. Il tag `<u>` è usato per sottolineare il testo.

Secondo la specifica HTML 5, l'elemento `<u>` dovrebbe essere usato solo quando non ci sono modi migliori per sottolineare il testo. I CSS forniscono un'alternativa moderna.

Font a Larghezza Fissa o Monospaziato

Per visualizzare il testo in un font *monospaziato* (a larghezza fissa), spesso usato per visualizzare il codice del computer, puoi usare l'elemento `<tt>` come illustrato nel seguente esempio. Il risultato appare in [Figure 14](#).

This `<tt>word</tt>` is in fixed-width font.

This word is in fixed-width font.

Figure 14. Il tag `<tt>` è usato per visualizzare il testo in un font a larghezza fissa.

Il tag `<tt>` non è supportato in HTML5. I browser web lo rendono ancora come previsto. Comunque, dovrresti usare tag più appropriati, che includono `<code>`, `<kbd>`, `<var>`, e `<samp>`.

Elementi di Frase

Gli elementi di frase non solo cambiano l'aspetto del testo, ma aggiungono anche importanza semantica a una parola o frase. Usandoli, si può enfatizzare una parola o marcarla come importante. Questi elementi, al contrario di presentazione, sono riconosciuti dagli screen reader, il che rende il testo più accessibile ai visitatori ipovedenti e permette ai motori di ricerca di leggere e valutare meglio il contenuto della pagina. Gli elementi di frase che usiamo in questa lezione sono ``, ``, e `<code>`.

Testo Enfatizzato

Per enfatizzare il testo, puoi inserirlo all'interno dell'elemento `` come illustrato nel seguente esempio:

This ``word`` is emphasized.

This **word** is emphasized.

Figure 15. Il tag `` è usato per enfatizzare il testo.

Come puoi vedere, i browser web visualizzano `` allo stesso modo di `<i>`, ma `` aggiunge importanza semantica come elemento di frase, il che migliora l'accessibilità per i visitatori ipovedenti.

Testo Rafforzato

Per marcare il testo come importante, puoi inserirlo dentro l'elemento `` come illustrato nell'esempio seguente. Il risultato appare in [Figure 16](#).

This ``word`` is important.

This **word** is important.

Figure 16. Il tag `` è usato per marcare il testo come importante.

Come puoi vedere, i browser web visualizzano `` allo stesso modo di ``, ma `` aggiunge importanza semantica come elemento di frase, il che migliora l'accessibilità per i visitatori ipovedenti.

Codice Informatico

Per inserire un pezzo di codice informatico, puoi inserirlo all'interno dell'elemento `<code>` come illustrato nell'esempio seguente. Il risultato appare in [Figure 17](#).

The Markdown code `<code># Heading</code>` creates a heading at the highest level in the hierarchy.

The Markdown code `# Heading` creates a heading at the highest level in the hierarchy.

Figure 17. Il tag `<code>` è usato per inserire un pezzo di codice informatico.

Testo Marcato

Per evidenziare il testo con uno sfondo giallo, simile allo stile di un evidenziatore, puoi usare l'elemento `<mark>` come illustrato nel seguente esempio. Il risultato appare in Figure 18.

This `<mark>word</mark>` is highlighted.

This word is highlighted.

Figure 18. Il tag `<mark>` è usato per evidenziare il testo con uno sfondo giallo.

Formattare il Testo della Nostra Lista della Spesa in HTML

Basandoci sui nostri esempi precedenti, inseriamo alcuni elementi di frase per cambiare l'aspetto del testo e allo stesso tempo aggiungere importanza semantica. Il risultato appare in Figure 19.

```
<h1>Garden Party</h1>
<p>
    Invitation to <strong>John's garden party</strong>. <br>
    <strong>Saturday, next week.</strong>
</p>

<h2>Agenda</h2>
<ol>
    <li>Welcome</li>
    <li>
        Barbecue
        <ul>
            <li><em>Vegetables</em></li>
            <li><em>Meat</em></li>
            <li><em>Burgers</em>, including vegetarian options</li>
        </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
</ol>

<hr>

<h2>Please bring</h2>
<ul>
    <li>Salad</li>
    <li>Drinks</li>
    <li>Bread</li>
    <li>Snacks</li>
    <li>Desserts</li>
</ul>
```

Garden Party

Invitation to John's garden party.
Saturday, next week.

Agenda

1. Welcome
 2. Barbecue
 - o *Vegetables*
 - o *Meat*
 - o *Burgers*, including vegetarian options
 3. Dessert
 4. **Fireworks**
-

Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

Figure 19. La pagina HTML con alcuni elementi di formattazione.

In questo documento HTML di esempio, le informazioni più importanti riguardanti la festa in giardino sono marcate come importanti usando l'elemento ``. I cibi che sono disponibili per il barbecue sono enfatizzati usando l'elemento ``. I fuochi d'artificio sono semplicemente evidenziati usando l'elemento `<mark>`.

Come esercizio, puoi provare a formattare altre parti di testo usando anche gli altri elementi di formattazione.

Testo Preformattato

Nella maggior parte degli elementi HTML, lo spazio bianco è solitamente ridotto a un singolo spazio o addirittura ignorato del tutto. Tuttavia, c'è un elemento HTML chiamato `<pre>` che ti permette di definire il cosiddetto testo *preformatto*. Lo spazio bianco nel contenuto di questo elemento, inclusi

gli spazi e le interruzioni di riga, viene conservato e reso nel browser web. Inoltre, il testo viene visualizzato in un font a larghezza fissa, simile all'elemento `<code>`.

```
<pre>
field() {
    shift $1 ; echo $1
}
</pre>
```

```
field() {
    shift $1 ; echo $1
}
```

Figure 20. Rappresentazione sul browser web del codice HTML che illustra come l'elemento HTML `<pre>` conserva lo spazio bianco.

Raggruppare gli Elementi

Per convenzione, gli elementi HTML sono divisi in due categorie:

Elementi Block-Level

Questi appaiono su una nuova linea e occupano l'intera larghezza disponibile. Esempi di elementi *block-level* che abbiamo già discusso sono `<p>`, ``, e `<h2>`.

Elementi Inline-Level

Questi appaiono nella stessa riga degli altri elementi e del testo, occupando solo lo spazio richiesto dal loro contenuto. Esempi di elementi di *inline-level* sono ``, ``, e `<i>`.

NOTE

HTML5 ha introdotto categorie di elementi più accurate e precise, cercando di evitare la confusione con i box CSS *block* e *inline*. Per semplicità, ci atterremo qui alla suddivisione convenzionale in elementi a blocchi e in linea.

Gli elementi fondamentali per raggruppare più elementi insieme sono gli elementi `<div>` e ``.

L'elemento `<div>` è un contenitore a livello di blocco per altri elementi HTML e non aggiunge valore semantico da solo. Puoi usare questo elemento per dividere un documento HTML in sezioni e strutturare il tuo contenuto, sia per la leggibilità del codice sia per applicare stili CSS a un gruppo di elementi, come imparerai in una lezione successiva.

Per impostazione predefinita, i browser web inseriscono sempre un'interruzione di riga prima e dopo ogni elemento `<div>` in modo che ognuno venga visualizzato sulla propria riga.

Al contrario, l'elemento `` è usato come contenitore per il testo HTML ed è generalmente usato per raggruppare altri elementi in linea al fine di applicare stili tramite CSS a una porzione più piccola di testo.

L'elemento `` si comporta proprio come il testo normale e non inizia su una nuova riga. È quindi un elemento inline.

L'esempio seguente confronta la rappresentazione visiva dell'elemento semantico `<p>` e degli elementi di raggruppamento `<div>` e ``:

```
<p>Text within a paragraph</p>
<p>Another paragraph of text</p>
<hr>
<div>Text wrapped within a <code>div</code> element</div>
<div>Another <code>div</code> element with more text</div>
<hr>
<span>Span content</span>
<span>and more span content</span>
```

Un browser web visualizza questo codice come mostrato in [Figure 21](#).

Text within a paragraph

Another paragraph of text

Text wrapped within a div element

Another div element with more text

Span content and more span content

Figure 21. Rappresentazione del browser web di un documento di prova che illustra le differenze tra gli elementi paragraph, div e span in HTML.

Abbiamo già visto che per default, il browser web aggiunge una spaziatura prima e dopo gli elementi `<p>`. Questa spaziatura non viene applicata a nessuno dei due elementi di raggruppamento `<div>` e ``. Tuttavia, gli elementi `<div>` sono formattati come blocchi propri, mentre il testo negli elementi `` è mostrato nella stessa riga.

Struttura della Pagina HTML

Abbiamo discusso come usare gli elementi HTML per descrivere semanticamente il contenuto di una pagina web: in altre parole, per trasmettere significato e contesto al testo. Un altro gruppo di elementi è progettato allo scopo di descrivere la *struttura semantica* di una pagina web, un'espressione o la sua struttura. Questi elementi sono elementi a blocchi, cioè si comportano visivamente in modo simile a un elemento `<div>`. Il loro scopo è quello di definire la struttura semantica di una pagina web specificando aree ben definite come intestazioni, piè di pagina e il contenuto principale della pagina. Questi elementi permettono il raggruppamento semantico del contenuto in modo che possa essere compreso anche da un computer, compresi i motori di ricerca e gli *screen reader*.

L'Elemento `<header>`

L'elemento `<header>` contiene informazioni introduttive all'elemento semantico circostante in un documento HTML. Un'intestazione è diversa da un titolo, ma un'intestazione spesso include un elemento di titolo (`<h1>`, ..., `<h6>`).

In pratica, questo elemento è usato più spesso per rappresentare l'intestazione della pagina, come un banner con un logo. Può anche essere usato per introdurre il contenuto di uno qualsiasi dei seguenti elementi: `<body>`, `<section>`, `<article>`, `<nav>`, o `<aside>`.

Un documento può avere più elementi `<header>`, ma un elemento `<header>` non può essere annidato dentro un altro elemento `<header>`. Né un elemento `<footer>` può essere usato all'interno di un `<header>`.

Per esempio, per aggiungere un'intestazione di pagina al nostro documento di test, possiamo fare quanto segue:

```
<header>
  <h1>Garden Party</h1>
</header>
```

Non ci saranno cambiamenti visibili al documento HTML, poiché `<h1>` (come tutti gli altri elementi di intestazione) è un elemento a livello di blocco senza altre proprietà visive.

L'Elemento di Contenuto `<main>`

L'elemento `<main>` è un contenitore per la parte centrale di una pagina web. Non ci deve essere più di un elemento `<main>` in un documento HTML.

Nel nostro documento d'esempio, tutto il codice HTML che abbiamo scritto finora verrebbe messo dentro l'elemento `<main>`.

```
<main>
  <header>
    <h1>Garden Party</h1>
  </header>
  <p>
    Invitation to <strong>John's garden party</strong>. <br>
    <strong>Saturday, next week.</strong>
  </p>

  <h2>Agenda</h2>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>

  <hr>

  <h2>Please bring</h2>
  <ul>
    <li>Salad</li>
    <li>Drinks</li>
    <li>Bread</li>
    <li>Snacks</li>
    <li>Desserts</li>
  </ul>
</main>
```

Come l'elemento `<header>`, l'elemento `<main>` non causa alcun cambiamento visivo nel nostro esempio.

L'Elemento <footer>

L'elemento `<footer>` contiene note a piè di pagina, per esempio informazioni sull'autore, informazioni di contatto, o documenti correlati, per il suo elemento semantico circostante, per esempio `<section>`, `<nav>`, o `<aside>`. Un documento può avere più elementi `<footer>` che permettono di descrivere meglio gli elementi semanticci. Comunque, un elemento `<footer>` non può essere annidato dentro un altro elemento `<footer>`, né un elemento `<header>` può essere usato dentro un `<footer>`.

Per il nostro esempio possiamo aggiungere informazioni di contatto (John) come mostrato nel seguente esempio.

```
<footer>
  <p>John Doe</p>
  <p>john.doe@example.com</p>
</footer>
```

L'Elemento <nav>

L'elemento `<nav>` descrive una rilevante unità di navigazione, come un menu, che contiene una serie di collegamenti ipertestuali.

NOTE

Non tutti i collegamenti ipertestuali devono essere contenuti in un elemento `<nav>`. È utile quando si elenca un gruppo di link.

Poiché i collegamenti ipertestuali non sono ancora stati trattati, l'elemento di navigazione non sarà incluso nell'esempio di questa lezione.

L'Elemento <aside>

L'elemento `<aside>` è un contenitore per ciò che non è necessario all'interno dell'ordine del contenuto principale della pagina, ma è di solito indirettamente collegato o supplementare. Questo elemento è spesso usato per barre laterali che mostrano informazioni secondarie, come un glossario.

Per il nostro esempio, possiamo aggiungere informazioni sull'indirizzo e sul viaggio, che sono solo indirettamente collegate al resto del contenuto, usando l'elemento `<aside>`.

```
<aside>
  <p>
    10, Main Street<br>
    Newville
  </p>
  <p>Parking spaces available.</p>
</aside>
```

L'Elemento `<section>`

L'elemento `<section>` definisce una sezione logica in un documento che è parte dell'elemento semantico circostante, ma non funzionerebbe come contenuto autonomo, come un capitolo.

Nel nostro documento di esempio, possiamo inserire le sezioni di contenuto per l'agenda e portare le sezioni di elenco come mostrato nel seguente esempio:

```

<section>
  <header>
    <h2>Agenda</h2>
  </header>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>
</section>

<hr>

<section>
  <header>
    <h2>Please bring</h2>
  </header>
  <ul>
    <li>Salad</li>
    <li>Drinks</li>
    <li>Bread</li>
    <li>Snacks</li>
    <li>Desserts</li>
  </ul>
</section>

```

Questo esempio aggiunge anche ulteriori elementi `<header>` all'interno delle sezioni, in modo che ogni sezione sia all'interno del proprio elemento `<header>`.

L'Elemento `<article>`

L'elemento `<article>` definisce un contenuto indipendente e autonomo che ha senso da solo senza il resto della pagina. Il suo contenuto è potenzialmente ridistribuibile o riutilizzabile in un altro contesto. Esempi tipici o materiale appropriato per un elemento `<article>` sono un post di un blog, un annuncio di un prodotto per un negozio o una pubblicità per un prodotto. La pubblicità potrebbe

quindi esistere sia da sola sia all'interno di una pagina più grande.

Nel nostro esempio, possiamo sostituire la prima `<section>` che avvolge l'agenda con un elemento `<article>`.

```
<article>
  <header>
    <h2>Agenda</h2>
  </header>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>
</article>
```

L'elemento `<header>` che abbiamo aggiunto nell'esempio precedente può persistere anche qui, perché gli elementi `<article>` possono avere i loro propri elementi `<header>`.

L'Esempio Finale

Combinando tutti gli esempi precedenti, il documento HTML finale per il tuo invito appare come segue:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Garden Party</title>
  </head>

  <body>
    <main>
      <h1>Garden Party</h1>
      <p>
```

```
Invitation to <strong>John's garden party</strong>. <br>
<strong>Saturday, next week.</strong>
</p>

<article>
  <h2>Agenda</h2>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>
</article>

<hr>

<section>
  <h2>Please bring</h2>
  <ul>
    <li>Salad</li>
    <li>Drinks</li>
    <li>Bread</li>
    <li>Snacks</li>
    <li>Desserts</li>
  </ul>
</section>
</main>

<aside>
  <p>
    10, Main Street<br>
    Newville
  </p>
  <p>Parking spaces available.</p>
</aside>

<footer>
  <p>John Doe</p>
```

```
<p>john.doe@example.com</p>
</footer>
</body>
</html>
```

In un browser web, l'intera pagina è visualizzata come mostrato in [Figure 22](#).

Garden Party

Invitation to **John's garden party**.
Saturday, next week.

Agenda

1. Welcome
 2. Barbecue
 - *Vegetables*
 - *Meat*
 - *Burgers*, including vegetarian options
 3. Dessert
 4. **Fireworks**
-

Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

10, Main Street
Newville

Parking spaces available.

John Doe

john.doe@example.com

Figure 22. Rappresentazione del browser web del documento HTML risultante che combina gli esempi precedenti. La pagina rappresenta un invito a una festa in giardino e descrive l'ordine del giorno della serata e una lista di cibo da portare per gli ospiti.

Esercizi Guidati

1. Per ognuno dei seguenti tag, indica il tag di chiusura corrispondente:

<h5>	
	
<dd>	
<hr>	
	
<tt>	
<main>	

2. Per ognuno dei seguenti tag, indica se segna l'inizio di un blocco o di un elemento *inline*:

<h3>	
	
	
<div>	
	
<dl>	
	
<nav>	
<code>	
<pre>	

3. Che tipo di liste puoi creare in HTML? Quali tag dovresti usare per ognuna di esse?

4. Quali tag racchiudono gli elementi di blocco che puoi usare per strutturare una pagina HTML?

Esercizi Esplorativi

1. Crea una pagina HTML di base con il titolo “Form Rules”. Userai questa pagina HTML per tutti gli esercizi esplorativi, ognuno dei quali è basato su quelli precedenti. Poi aggiungi un’intestazione di livello 1 con il testo “How to fill in the request form”, un paragrafo con il testo “To receive the PDF document with the complete HTML course, it is necessary to fill in the following fields:” e una lista non ordinata con i seguenti elementi della lista: “Name”, “Surname”, “Email Address”, “Nation”, “Country”, e “Zip/Postal Code”.

2. Metti i primi tre campi (“Name”, “Surname”, and “Email Address”) in grassetto, aggiungendo anche un’importanza semantica. Poi aggiungi un’intestazione di livello 2 con il testo “Required fields” e un paragrafo con il testo “Bold fields are mandatory.”

3. Aggiungi un altro titolo di livello 2 con il testo “Steps to follow”, un paragrafo con il testo “There are four steps to follow:”, e una lista ordinata con le seguenti voci: “Fill in the fields”, “Click the Submit button”, “Check your e-mail and confirm your request by clicking on the link you receive”, e “Check your e-mail - You will receive the full HTML course in minutes”.

4. Usando `<div>`, crea un blocco per ogni sezione che inizia con un’intestazione di livello 2.

5. Usando `<div>`, crea un altro blocco per la sezione che inizia con il titolo del livello 1. Poi dividi questa sezione dalle altre due con una linea orizzontale.

6. Aggiungi l’elemento di intestazione con il testo “Form Rules - 2021” e l’elemento di piè di pagina con il testo “Copyright Note - 2021”. Infine, aggiungi l’elemento principale che deve contenere i tre blocchi `<div>`.

Sommario

In questa lezione hai imparato:

- Come creare *markup* per i contenuti in un documento HTML
- La struttura gerarchica del testo HTML
- La differenza tra elementi HTML a blocchi e in linea
- Come creare documenti HTML con una struttura semantica

I seguenti termini sono stati discussi in questa lezione:

<h1>, <h2>, <h3>, <h4>, <h5>, <h6>

I tag di intestazione.

<p>

Il tag paragrafo.

Il tag della lista ordinata.

Il tag della lista non ordinata.

Il tag dell'elemento della lista.

<dl>

Il tag descrizione lista.

<dt>, <dd>

I tag di ogni termine e descrizione per una lista di descrizione.

<pre>

Il tag di mantenimento della formattazione.

, <i>, <u>, <tt>, , , <code>, <mark>

I tag di formattazione.

**<div>, **

I tag di raggruppamento.

<header>, <main>, <nav>, <aside>, <footer>

I tag utilizzati per fornire una struttura e un layout semplici a una pagina HTML.

Risposte agli Esercizi Guidati

1. Per ognuno dei seguenti tag, indica il tag di chiusura corrispondente:

<h5>	</h5>
 	Non esiste
	
<dd>	</dd>
<hr>	non esiste
	
<tt>	</tt>
<main>	</main>

2. Per ognuno dei seguenti tag, indica se segna l'inizio di un blocco o di un elemento *inline*:

<h3>	Elemento Blocco
	Elemento Inline
	Elemento Inline
<div>	Elemento Blocco
	Elemento Inline
<dl>	Elemento Blocco
	Elemento Blocco
<nav>	Elemento Blocco
<code>	Elemento Inline
<pre>	Elemento Blocco

3. Che tipo di liste puoi creare in HTML? Quali tag dovresti usare per ognuno di essi?

In HTML, puoi creare tre tipi di elenchi: elenchi ordinati costituiti da una serie di elementi di elenco numerati, elenchi non ordinati costituiti da una serie di elementi di un elenco che non hanno un ordine o una sequenza speciale ed elenchi descrittivi che rappresentano voci come in un dizionario o in un'encyclopedia. Un elenco ordinato è racchiuso tra i tag e , un elenco non ordinato è racchiuso tra i tag e e un elenco di descrizioni è racchiuso tra i tag <dl> e </dl>. Ogni elemento in un elenco ordinato o non ordinato è racchiuso tra i tag e , mentre ogni termine in un elenco descrittivo è racchiuso tra i tag <dt> e </dt> e la

sua descrizione è racchiusa tra i tag `<dd>` e `</dd>`.

4. Quali tag racchiudono gli elementi di blocco che puoi usare per strutturare una pagina HTML?

I tag `<header>` e `</header>` racchiudono l'intestazione della pagina, i tag `<main>` e `</main>` racchiudono il contenuto principale della pagina HTML, i tag `<nav>` e `</nav>` racchiudono la cosiddetta barra di navigazione, i tag `<aside>` e `</aside>` racchiudono la barra laterale e i tag `<footer>` e `</footer>` racchiudono la pagina più di pagina.

Risposte agli Esercizi Esplorativi

- Crea una pagina HTML di base con il titolo “Form Rules”. Userai questa pagina HTML per tutti gli esercizi esplorativi, ognuno dei quali è basato su quelli precedenti. Poi aggiungi un’intestazione di livello 1 con il testo “How to fill in the request form”, un paragrafo con il testo “To receive the PDF document with the complete HTML course, it is necessary to fill in the following fields:” e una lista non ordinata con i seguenti elementi della lista: “Name”, “Surname”, “Email Address”, “Nation”, “Country”, e “Zip/Postal Code”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li>Name</li>
      <li>Surname</li>
      <li>Email Address</li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>
  </body>
</html>
```

- Metti i primi tre campi (“Name”, “Surname”, and “Email Address”) in grassetto, aggiungendo anche un’importanza semantica. Poi aggiungi un’intestazione di livello 2 con il testo “Required fields” e un paragrafo con il testo “Bold fields are mandatory.”

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
      <li><strong> Email Address </strong></li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>

    <h2>Required fields</h2>
    <p>Bold fields are mandatory.</p>
  </body>
</html>
```

3. Aggiungi un altro titolo di livello 2 con il testo “Steps to follow”, un paragrafo con il testo “There are four steps to follow:”, e una lista ordinata con le seguenti voci: “Fill in the fields”, “Click the Submit button”, “Check your e-mail and confirm your request by clicking on the link you receive”, e “Check your e-mail - You will receive the full HTML course in minutes”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
      <li><strong> Email Address </strong></li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>

    <h2>Required fields</h2>
    <p>Bold fields are mandatory.</p>

    <h2>Steps to follow</h2>
    <p>There are four steps to follow:</p>
    <ol>
      <li>Fill in the fields</li>
      <li>Click the Submit button</li>
      <li>
        Check your e-mail and confirm your request by clicking on the link you
        receive
      </li>
      <li>
        Check your e-mail – You will receive the full HTML course in minutes
      </li>
    </ol>
  </body>
</html>
```

4. Usando `<div>`, crea un blocco per ogni sezione che inizia con un titolo di livello 2.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
      <li><strong> Email Address </strong></li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>

    <div>
      <h2>Required fields</h2>
      <p>Bold fields are mandatory.</p>
    </div>

    <div>
      <h2>Steps to follow</h2>
      <p>There are four steps to follow:</p>
      <ol>
        <li>Fill in the fields</li>
        <li>Click the Submit button</li>
        <li>
          Check your e-mail and confirm your request by clicking on the link
          you
          receive
        </li>
        <li>
          Check your e-mail – You will receive the full HTML course in minutes
        </li>
      </ol>
    </div>
  </body>
</html>
```

5. Usando <div>, crea un altro blocco per la sezione che inizia con un titolo di livello 1. Poi dividi questa sezione dalle altre due con una linea orizzontale.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <div>
      <h1>How to fill in the request form</h1>
      <p>
        To receive the PDF document with the complete HTML course, it is
        necessary to fill in the following fields:
      </p>
      <ul>
        <li><strong> Name </strong></li>
        <li><strong> Surname </strong></li>
        <li><strong> Email Address </strong></li>
        <li>Nation</li>
        <li>Country</li>
        <li>Zip/Postal Code</li>
      </ul>
    </div>

    <hr>

    <div>
      <h2>Required fields</h2>
      <p>Bold fields are mandatory.</p>
    </div>

    <div>
      <h2>Steps to follow</h2>
      <p>There are four steps to follow:</p>
      <ol>
        <li>Fill in the fields</li>
        <li>Click the Submit button</li>
        <li>
          Check your e-mail and confirm your request by clicking on the link
          you
          receive
        </li>
      </ol>
    </div>
  </body>
</html>
```

```

<li>
    Check your e-mail – You will receive the full HTML course in minutes
</li>
</ol>
</div>
</body>
</html>

```

6. Aggiungi l'elemento di intestazione con il testo "Form Rules - 2021" e l'elemento di piè di pagina con il testo "Copyright Note - 2021". Infine, aggiungi l'elemento principale che deve contenere i tre blocchi <div>.

```

<!DOCTYPE html>
<html>
    <head>
        <title>Form Rules</title>
    </head>

    <body>
        <header>
            <h1>Form Rules – 2021</h1>
        </header>

        <main>
            <div>
                <h1>How to fill in the request form</h1>
                <p>
                    To receive the PDF document with the complete HTML course, it is
                    necessary to fill in the following fields:
                </p>
                <ul>
                    <li><strong> Name </strong></li>
                    <li><strong> Surname </strong></li>
                    <li><strong> Email Address </strong></li>
                    <li>Nation</li>
                    <li>Country</li>
                    <li>Zip/Postal Code</li>
                </ul>
            </div>

            <hr>

            <div>
                <h2>Required fields</h2>

```

```
<p>Bold fields are mandatory.</p>
</div>

<div>
  <h2>Steps to follow</h2>
  <p>There are four steps to follow:</p>
  <ol>
    <li>Fill in the fields</li>
    <li>Click the Submit button</li>
    <li>
      Check your e-mail and confirm your request by clicking on the link
      you receive
    </li>
    <li>
      Check your e-mail – You will receive the full HTML course in
      minutes
    </li>
  </ol>
</div>
</main>

<footer>
  <p>Copyright Note – 2021</p>
</footer>
</body>
</html>
```



032.3 Riferimenti HTML e Risorse Integrate

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 032.3

Peso

2

Arearie di Conoscenza Chiave

- Creare semplici moduli HTML
- Comprendere i metodi dei moduli HTML
- Comprendere gli elementi e i tipi di input HTML

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- Attributo `id`
- `<a>`, inclusi gli attributi `href` e `target` (`_blank`, `_self`, `_parent`, `_top`)
- ``, inclusi gli attributi `src` e `alt`



032.3 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	032 Marcatura di un Documento HTML
Obiettivo:	032.3 Riferimenti HTML e Risorse Integrate
Lezione:	1 di 1

Introduzione

Qualsiasi pagina web moderna è raramente costituita da solo testo. Comprende molti altri tipi di contenuti, come immagini, audio, video e anche altri documenti HTML. Insieme al contenuto esterno, i documenti HTML possono contenere link ad altri documenti, il che rende l'esperienza di navigazione Internet molto più semplice.

Contenuto Integrato

Lo scambio di file è possibile su Internet senza pagine web scritte in HTML, quindi perché l'HTML è il formato scelto per i documenti web, e non il PDF o qualsiasi altro formato di elaborazione testi? Una ragione importante è che l'HTML mantiene le sue risorse multimediali in file *separati*. In un ambiente come Internet, dove le informazioni sono spesso ridondanti e distribuite in luoghi diversi, è importante evitare trasferimenti di dati non necessari. La maggior parte delle volte, le nuove versioni di una pagina web contengono le stesse immagini e altri file di supporto delle versioni precedenti, così il browser web può usare i file precedentemente recuperati invece di copiare tutto di nuovo. Inoltre, mantenere i file separati facilita la personalizzazione del contenuto multimediale secondo le caratteristiche del cliente, come la sua posizione, la dimensione dello schermo e la velocità di connessione.

Immagini

Le immagini sono il tipo più comune di contenuto incorporato ad accompagnare il testo. Le immagini sono conservate separatamente e sono referenziate all'interno del file HTML con il tag ``:

```

```

Il tag `` non richiede un tag di chiusura. La proprietà `src` indica la posizione di origine del file immagine. In questo esempio, il file immagine `logo.png` deve trovarsi nella stessa directory del file HTML, altrimenti il browser non sarà in grado di visualizzarlo. La proprietà `source location` accetta percorsi relativi, quindi la *notazione puntata* può essere usata per indicare il percorso dell'immagine:

```

```

I due punti indicano che l'immagine si trova all'interno della directory contenitore relativa alla directory dove si trova il file HTML. Se il nome del file `../logo.png` è usato dentro un file HTML il cui URL è `http://example.com/library/periodicals/index.html`, il browser richiederà il file immagine all'indirizzo `http://example.com/library/logo.png`.

La notazione puntata si applica anche se il file HTML non è un file reale nel filesystem; il browser HTML interpreta l'URL come se fosse un percorso a un file, ma è compito del server HTTP decidere se quel percorso si riferisce a un file o a un contenuto generato dinamicamente. Il dominio e il percorso corretto vengono aggiunti automaticamente a tutte le richieste al server, nel caso in cui il file HTML provenga da una richiesta HTTP. Allo stesso modo, il browser aprirà l'immagine corretta se il file HTML è stato aperto direttamente dal filesystem locale.

I percorsi di origine che iniziano con uno slash / sono trattati come percorsi assoluti. I percorsi assoluti hanno informazioni complete per le posizioni dell'immagine, quindi funzionano indipendentemente dalla posizione del documento HTML. Se il file dell'immagine si trova su un altro server, che sarà il caso quando viene usata un *Content Delivery Network* (CDN), deve essere incluso anche il nome del dominio.

NOTE

I Content Delivery Network sono composte da server geograficamente distribuiti che immagazzinano contenuti statici per altri siti web. Aiutano a migliorare le prestazioni e la disponibilità per i siti a forte accesso.

Se l'immagine non può essere caricata, il browser HTML mostrerà il testo fornito dall'attributo `alt` invece dell'immagine. Per esempio:

```

```

L'attributo `alt` è anche importante per l'accessibilità. I browser "di solo testo" e gli screen reader lo usano come descrizione dell'immagine corrispondente.

Tipi d'Immagine

I browser web possono visualizzare *tutti* i tipi di immagini più popolari, come JPEG, PNG, GIF e SVG. Le dimensioni delle immagini vengono rilevate non appena le immagini vengono caricate, ma possono essere predefinite con gli attributi `width` e `height`:

```

```

L'unica ragione per includere gli attributi di dimensione al tag `` è di evitare di rovinare il layout di pagina quando l'immagine impiega troppo tempo per essere caricata o quando non può essere caricata affatto. Usare gli attributi `width` e `height` per cambiare le dimensioni originali dell'immagine può portare a risultati indesiderati:

- Le immagini saranno distorte quando la dimensione originale è più piccola delle nuove dimensioni o quando il nuovo rapporto di proporzioni è diverso dall'originale.
- Il ridimensionamento delle immagini di grandi dimensioni utilizza una larghezza di banda extra che si traduce in tempi di caricamento più lunghi.

SVG è l'unico formato che non soffre di questi effetti, perché tutte le sue informazioni grafiche sono memorizzate in coordinate numeriche adatte al ridimensionamento e le sue dimensioni non influenzano la dimensione del file (da qui il nome *Scalable Vector Graphics*). Per esempio, solo la posizione, le dimensioni laterali e le informazioni sul colore sono necessarie per disegnare un rettangolo in SVG. Il valore particolare per ogni singolo pixel sarà reso dinamicamente in seguito. In effetti, le immagini SVG sono simili ai file HTML, nel senso che i loro elementi grafici sono anche definiti da tag in un file di testo. I file SVG sono destinati a rappresentare disegni a spigoli vivi, come grafici o diagrammi.

Le immagini che non rientrano in questi criteri dovrebbero essere memorizzate come *bitmap*. A differenza dei formati immagine basati su vettori, i bitmap memorizzano in anticipo le informazioni sul colore per ogni pixel dell'immagine. Memorizzare il valore del colore per ogni pixel dell'immagine genera una grande quantità di dati, quindi le bitmap sono solitamente memorizzate in formati compressi, come JPEG, PNG o GIF.

Il formato JPEG è raccomandato per le fotografie, perché il suo algoritmo di compressione produce

buoni risultati per le sfumature e gli sfondi sfocati. Per le immagini in cui prevalgono i colori solidi, il formato PNG è più appropriato. Pertanto, il formato PNG dovrebbe essere scelto quando è necessario convertire un'immagine vettoriale in una bitmap.

Il formato GIF offre la qualità d'immagine più bassa di tutti i formati bitmap popolari. Tuttavia, è ancora ampiamente utilizzato a causa del suo supporto alle animazioni. Molti siti web utilizzano i file GIF per visualizzare brevi video, ma vedremo come ci siano modi migliori per visualizzare questo tipo di contenuto.

Audio e Video

I contenuti audio e video possono essere aggiunti a un documento HTML più o meno allo stesso modo delle immagini. Non sorprende che il tag per aggiungere audio sia `<audio>` e quello per aggiungere video sia `<video>`. Ovviamente, i browser di solo testo non sono in grado di riprodurre contenuti multimediali, quindi i tag `<audio>` e `<video>` impiegano il tag di chiusura per contenere il testo usato come fallback per l'elemento che non può essere mostrato. Per esempio:

```
<audio controls src="/media/recording.mp3">
<p>Unable to play <em>recording.mp3</em></p>
</audio>
```

Se il browser non supporta il tag `<audio>`, verrà invece mostrata la riga “Unable to play recording.mp3”. L'uso dei tag di chiusura `</audio>` o `</video>` permette a una pagina web di includere contenuti alternativi più elaborati rispetto alla semplice linea di testo permessa dall'attributo `alt` del tag ``.

L'attributo `src` per i tag `<audio>` e `<video>` funziona allo stesso modo del tag ``, ma accetta anche URL che puntano a un flusso live. Il browser si occupa di bufferizzare, decodificare e visualizzare il contenuto come viene ricevuto. L'attributo `controls` visualizza i controlli di riproduzione. Senza di esso, il visitatore non sarà in grado di mettere in pausa, riavvolgere o controllare in altro modo la riproduzione.

Il Contenuto Generico

Un documento HTML può essere annidato in un altro documento HTML, in modo simile all'inserimento di un'immagine in un documento HTML, ma usando il tag `<iframe>`:

```
<iframe name="viewer" src="gallery.html">
<p>Unsupported browser</p>
</iframe>
```

I browser più semplici di solo testo *non* supportano il tag `<iframe>` e visualizzeranno invece il testo racchiuso. Come per i tag multimediali, l'attributo `src` imposta la posizione sorgente del documento annidato. Gli attributi `width` e `height` possono essere aggiunti per cambiare le dimensioni predefinite dell'elemento `iframe`.

L'attributo `name` permette di fare riferimento all'iframe e di cambiare il documento annidato. Senza questo attributo, il documento annidato non può essere cambiato. Un elemento `anchor` può essere usato per caricare un documento da un'altra posizione all'interno di un iframe invece che dalla finestra corrente del browser.

Collegamenti

L'elemento della pagina comunemente indicato come *link* web è anche conosciuto con il termine tecnico *anchor*, da cui l'uso del tag `<a>`. Il collegamento (link) conduce a un altro “luogo”, che può essere qualsiasi indirizzo supportato dal browser. La destinazione è indicata dall'attributo `href` (*hyperlink reference*):

```
<a href="contact.html">Contact Information</a>
```

La posizione può essere scritta come un percorso relativo o assoluto, come per i contenuti incorporati discussi in precedenza. Solo il contenuto testuale racchiuso (per esempio, Contact Information) è visibile al visitatore, di solito per impostazione predefinita come testo blu sottolineato cliccabile, ma l'elemento visualizzato sopra il link può anche essere qualsiasi altro contenuto visibile, come le immagini:

```
<a href="contact.html"></a>
```

Si possono aggiungere prefissi speciali alla posizione per dire al browser come aprirla. Se il riferimento punta a un indirizzo email, per esempio, il suo attributo `href` dovrebbe includere il prefisso `mailto::`:

```
<a href="mailto:info@lpi.org">Contact by email</a>
```

Il prefisso `tel:` indica un numero di telefono. È particolarmente utile per i visitatori che visualizzano la pagina su dispositivi mobili:

```
<a href="tel:+123456789">Contact by phone</a>
```

Quando il link viene cliccato, il browser apre il contenuto indicato con l'applicazione associata.

L'uso più comune dei link è quello di caricare altri documenti web. Per impostazione predefinita, il browser sostituirà il documento HTML corrente con il contenuto nella nuova posizione. Questo comportamento può essere modificato usando l'attributo `target`. L'obiettivo `_blank`, per esempio, dice al browser di aprire la posizione data in una nuova finestra o in una nuova scheda del browser, a seconda delle preferenze del visitatore:

```
<a href="contact.html" target="_blank">Contact Information</a>
```

Il target `_self` è quello predefinito quando l'attributo `target` non è fornito. Fa sì che il documento referenziato sostituisca il documento corrente.

Altri tipi di target sono legati all'elemento `<iframe>`. Per caricare un documento di riferimento all'interno di un elemento `<iframe>`, l'attributo `target` dovrebbe puntare al nome dell'elemento `iframe`:

```
<p><a href="gallery.html" target="viewer">Photo Gallery</a></p>

<iframe name="viewer" width="800" height="600">
<p>Unsupported browser</p>
</iframe>
```

L'elemento `iframe` funziona come una finestra distinta del browser, quindi qualsiasi link caricato dal documento dentro l'`iframe` sostituirà solo il contenuto dell'`iframe`. Per cambiare questo comportamento, gli elementi di ancoraggio all'interno del documento incorniciato possono anche usare l'attributo `target`. Il target `_parent`, quando usato all'interno di un documento incorniciato, farà sì che la posizione di riferimento sostituisca il documento padre contenente il tag `<iframe>`. Per esempio, il documento incorporato `gallery.html` potrebbe contenere un link che carica se stesso mentre sostituisce il documento padre:

```
<p><a href="gallery.html" target="_parent">Open as parent document</a></p>
```

I documenti HTML supportano livelli multipli di annidamento con il tag `<iframe>`. Il target `_top`, quando usato in un'ancora all'interno di un documento incorniciato, farà sì che la posizione di riferimento sostituisca il documento principale nella finestra del browser, indipendentemente dal fatto che sia il genitore immediato del corrispondente `<iframe>` o un "antenato" più indietro nella catena.

Posizioni all'Interno dei Documenti

L'indirizzo di un documento HTML può opzionalmente contenere un *frammento* che può essere usato per identificare una risorsa all'interno del documento. Questo frammento, noto anche come *URL anchor*, è una stringa che segue un carattere di hash # alla fine dell'URL. Per esempio, la parola History è una *anchor* nella URL <https://en.wikipedia.org/wiki/Internet#History>.

Quando l'URL ha un frammento, il browser scorrerà fino all'elemento corrispondente nel documento: cioè l'elemento il cui attributo id è uguale alla URL anchor. Nel caso dell'URL dato, <https://en.wikipedia.org/wiki/Internet#History>, il browser salterà direttamente alla sezione "History". Esaminando il codice HTML della pagina, scopriamo che il titolo della sezione ha l'attributo id corrispondente:

```
<span class="mw-headline" id="History">History</span>
```

Le *URL anchor* possono essere usate nell'attributo href del tag `<a>`, sia quando puntano a pagine esterne sia quando puntano a posizioni all'interno della pagina corrente. In quest'ultimo caso, è sufficiente iniziare con il solo segno di *hash* con il frammento di URL, come in `History`.

WARNING

L'attributo id non deve contenere spazi (spazi, tabulazioni, ecc.) e deve essere unico all'interno del documento.

Ci sono modi per personalizzare il modo in cui il browser reagisce ai frammenti di URL. È possibile, per esempio, scrivere una funzione JavaScript che recepisca l'evento *hashchange* della finestra e inneschi un'azione personalizzata, come un'animazione o una richiesta HTTP. Vale la pena notare, tuttavia, che il frammento di URL non viene mai inviato al server con l'URL, quindi non può essere utilizzato come identificatore dal server HTTP.

Esercizi Guidati

- Il documento HTML situato in <http://www.lpi.org/articles/linux/index.html> ha un tag `` il cui attributo `src` punta a `../logo.png`. Qual è il percorso assoluto completo di questa immagine?

- Indica due ragioni per cui l'attributo `alt` è importante nei tag ``.

- Quale formato d'immagine dà una buona qualità e mantiene la dimensione del file contenuta quando è usato per fotografie con punti sfocati e con molti colori e sfumature?

- Invece di usare un provider di terze parti come Youtube, quale tag HTML ti permette di incorporare un file video in un documento HTML usando solo le caratteristiche standard dell'HTML?

Esercizi Esplorativi

- Supponiamo che un documento HTML abbia il collegamento ipertestuale `First picture` e l'elemento iframe `<iframe name="gallery"></iframe>`. Come potresti modificare il tag hyperlink in modo che l'immagine a cui punta venga caricata all'interno dell'elemento iframe dopo che l'utente ha cliccato sul link?

- Che cosa succede quando il visitatore clicca su un collegamento ipertestuale in un documento all'interno di un iframe e il collegamento ipertestuale ha l'attributo target impostato su `_self`?

- Noti che l'anchor URL per la seconda sezione della tua pagina HTML non funziona. Qual è la probabile causa di questo errore?

Sommario

Questa lezione tratta del come aggiungere immagini e altri contenuti multimediali usando i tag HTML appropriati. Inoltre, il lettore impara i diversi modi in cui i collegamenti ipertestuali possono essere usati per caricare altri documenti e puntare a posizioni specifiche all'interno di una pagina. La lezione si occupa dei seguenti concetti e procedure:

- Il tag `` e i suoi attributi principali: `src` e `alt`.
- Percorsi URL relativi e assoluti.
- I formati immagine più diffusi per il Web e le loro caratteristiche.
- I tag multimediali `<audio>` e `<video>`.
- Come inserire documenti annidati con il tag `<iframe>`.
- Il tag hyperlink `<a>`, il suo attributo `href`, e gli target speciali.
- Come usare i frammenti di URL, conosciuti anche come *hash anchors*.

Risposte agli Esercizi Guidati

- Il documento HTML situato in <http://www.lpi.org/articles/linux/index.html> ha un tag `` il cui attributo `src` punta a `../logo.png`. Qual è il percorso assoluto completo di questa immagine?

`http://www.lpi.org/articles/logo.png`

- Indica due ragioni per cui l'attributo `alt` è importante nei tag ``.

I browser di solo testo saranno in grado di mostrare una descrizione dell'immagine mancante. Gli screen reader usano l'attributo `alt` per descrivere l'immagine.

- Quale formato d'immagine dà una buona qualità e mantiene la dimensione del file contenuta quando è usato per fotografie con punti sfocati e con molti colori e sfumature?

Il formato JPEG.

- Invece di usare un provider di terze parti come Youtube, quale tag HTML ti permette di incorporare un file video in un documento HTML usando solo le caratteristiche standard dell'HTML?

Il tag `<video>`.

Risposte agli Esercizi Esplorativi

- Supponiamo che un documento HTML abbia il collegamento ipertestuale `First picture` e l'elemento iframe `<iframe name="gallery"></iframe>`. Come potresti modificare il tag hyperlink in modo che l'immagine a cui punta venga caricata all'interno dell'elemento iframe dopo che l'utente ha cliccato sul link?

Usando l'attributo `target` del tag `a`: `First picture`.

- Che cosa succede quando il visitatore clicca su un collegamento ipertestuale in un documento all'interno di un iframe e il collegamento ipertestuale ha l'attributo `target` impostato su `_self`?

Il documento sarà caricato all'interno dello stesso iframe, che è il comportamento predefinito.

- Noti che l'anchor URL per la seconda sezione della tua pagina HTML non funziona. Qual è la probabile causa di questo errore?

Il frammento di URL dopo il segno di hash non corrisponde all'attributo `id` nell'elemento corrispondente alla seconda sezione, oppure l'attributo `id` dell'elemento non è presente.



032.4 Moduli HTML

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 032.4

Peso

2

Arese di Conoscenza Chiave

- Creare link a risorse esterne e collegamenti di pagina
- Aggiungere immagini ai documenti HTML
- Comprendere le proprietà chiave dei comuni formati di file multimediali, inclusi PNG, JPG e SVG.
- Conoscenza degli iframe

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- `<form>`, inclusi gli attributi `method` (get, post), `action` e `enctype`
- `<input>`, inclusi gli attributi `type` (text, email, password, number, date, file, range, radio, checkbox, hidden)
- `<button>`, incluso l'attributo `type` (submit, reset, hidden)
- `<textarea>`
- Comuni attributi degli elementi dei moduli (`name`, `value`, `id`)
- `<label>`, incluso l'attributo `for`



032.4 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	032 Marcatura di un Documento HTML
Obiettivo:	032.4 Moduli HTML
Lezione:	1 di 1

Introduzione

I moduli (*form*) web forniscono un modo semplice ed efficiente per richiedere informazioni ai visitatori su una pagina HTML. Lo sviluppatore front-end può utilizzare vari componenti come campi di testo, caselle di controllo, pulsanti e molti altro per “costruire” interfacce che invieranno dati al server in modo strutturato.

Semplici Moduli HTML

Prima di vedere nel dettaglio il codice di markup specifico per i moduli, iniziamo con un semplice documento HTML vuoto, senza alcun contenuto del corpo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with HTML Forms</title>
</head>
<body>

<!-- The body content goes here --&gt;

&lt;/body&gt;
&lt;/html&gt;</pre>
```

Salva l'esempio di codice come file di testo semplice con estensione .html (come in form.html) e usa il tuo browser preferito per aprirlo. Dopo averlo modificato, premi il pulsante reload nel browser per mostrare le modifiche.

La struttura di base del modulo è data dal tag `<form>` stesso e dai suoi elementi interni:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with HTML Forms</title>
</head>
<body>

<!-- Form to collect personal information --&gt;

&lt;form&gt;

&lt;h2&gt;Personal Information&lt;/h2&gt;

&lt;p&gt;Full name:&lt;/p&gt;
&lt;p&gt;&lt;input type="text" name="fullname" id="fullname"&gt;&lt;/p&gt;

&lt;p&gt;&lt;input type="reset" value="Clear form"&gt;&lt;/p&gt;
&lt;p&gt;&lt;input type="submit" value="Submit form"&gt;&lt;/p&gt;

&lt;/form&gt;

&lt;/body&gt;
&lt;/html&gt;</pre>
```

I doppi apici non sono richiesti per gli attributi a parola singola come `type`, quindi `type=text` funziona bene come `type="text"`. Lo sviluppatore può scegliere quale convenzione utilizzare.

Salva il nuovo contenuto e ricarica la pagina nel browser. Dovresti vedere il risultato mostrato in [Figure 23](#).

Personal Information

Full name:

[Clear form](#)

[Submit form](#)

Figure 23. Un form molto elementare.

Il tag `<form>` da solo non produce alcun risultato evidente sulla pagina. Gli elementi all'interno dei tag `<form>...</form>` definiranno i campi e altri aiuti visivi mostrati al visitatore.

Il codice di esempio contiene entrambi i tag HTML generici (`<h2>` e `<p>`) e il tag `<input>`, che è un tag specifico del modulo. Mentre i tag generici possono apparire ovunque nel documento, i tag specifici del modulo dovrebbero essere usati solo all'interno dell'elemento `<form>`; cioè tra i tag di apertura `<form>` e di chiusura `</form>`.

NOTE

L'HTML fornisce solo tag e proprietà di base per modificare l'aspetto standard dei moduli. I CSS forniscono meccanismi elaborati per modificare l'aspetto del modulo, quindi la raccomandazione è quella di scrivere codice HTML che si occupa solo degli aspetti funzionali del modulo e modificare il suo aspetto attraverso i CSS.

Come mostrato nell'esempio, il tag paragrafo `<p>` può essere usato per descrivere il campo al visitatore. Tuttavia, non c'è un modo ovvio in cui il browser possa mettere in relazione la descrizione nel tag `<p>` con l'elemento di input corrispondente. Il tag `<label>` è più appropriato in questi casi (d'ora in poi, considera tutti gli esempi di codice come se fossero all'interno del corpo del documento HTML):

```
<form>
```

```
  <h2>Personal Information</h2>
```

```
  <label for="fullname">Full name:</label>
  <p><input type="text" name="fullname" id="fullname"></p>

  <p><input type="reset" value="Clear form"></p>
  <p><input type="submit" value="Submit form"></p>

</form>
```

L'attributo `for` nel tag `<label>` contiene l' `id` del corrispondente elemento di input. Rende la pagina più accessibile, poiché gli *screenreader* saranno in grado di pronunciare il contenuto dell'elemento etichetta quando l'elemento di input è evidenziato. Inoltre, i visitatori possono cliccare sull'etichetta per mettere a fuoco il campo di input corrispondente.

L'attributo `id` funziona per gli elementi del modulo come per qualsiasi altro elemento nel documento. Fornisce un identificatore per l'elemento che è unico all'interno dell'intero documento. L'attributo `name` ha uno scopo simile, ma è usato per identificare l'elemento di input nel contesto del modulo. Il browser usa l'attributo `name` per identificare il campo di input quando invia i dati del modulo al server, quindi è importante usare attributi `name` significativi e unici all'interno del modulo.

L'attributo `type` è l'attributo principale dell'elemento `input`, perché controlla il tipo di dati che l'elemento accetta e la sua presentazione visiva al visitatore. Se l'attributo `type` non è fornito, per default l'input mostra una casella di testo. I seguenti tipi di input sono supportati dai browser moderni:

Table 1. Tipi di input form

Tipo di attributo	Tipo di dati	Come viene visualizzato
<code>hidden</code>	Una stringa arbitraria	N/A
<code>text</code>	Testo senza interruzioni di riga	Un controllo di testo
<code>search</code>	Testo senza interruzioni di riga	Un controllo di ricerca
<code>tel</code>	Testo senza interruzioni di riga	Un controllo di testo
<code>url</code>	Un URL assoluto	Un controllo di testo
<code>email</code>	Un indirizzo email o una lista di indirizzi email	Un controllo di testo

Tipo di attributo	Tipo di dati	Come viene visualizzato
password	Testo senza interruzioni di riga (informazioni sensibili)	Un controllo di testo che oscura l'inserimento dei dati
date	Una data (anno, mese, giorno) senza fuso orario	Un controllo di data
month	Una data che consiste di un anno e un mese senza fuso orario	Un controllo del mese
week	Una data che consiste in un numero di settimana-anno e un numero di settimana senza fuso orario	Un controllo di settimana
time	Un'ora (ora, minuto, secondo, frazione di secondo) senza fuso orario	Un controllo orario
datetime-local	Una data e un'ora (anno, mese, giorno, ora, minuto, secondo, frazione di secondo) senza fuso orario	Un controllo di data e ora
number	Un valore numerico	Un controllo testo o un controllo <i>spinner</i>
range	Un valore numerico, con la semantica extra	Un controllo <i>slider</i> o simile
color	Un colore sRGB con componenti rosso, verde e blu a 8 bit	Un selezionatore di colori
checkbox	Un insieme di zero o più valori da una lista predefinita	Una casella di controllo (offre scelte e permette la selezione di scelte multiple)
radio	Un valore enumerato	Un pulsante radio (offre più scelte e permette la selezione di una sola scelta)
file	Zero o più file, ciascuno con un tipo MIME e un nome di file opzionale	Un'etichetta e un pulsante

Tipo di attributo	Tipo di dati	Come viene visualizzato
submit	Un valore enumerato, che termina il processo di input e fa sì che il modulo venga inviato	Un pulsante
image	Una coordinata, relativa a una particolare dimensione dell'immagine, che termina il processo di inserimento e fa sì che il modulo venga inviato	Un'immagine cliccabile o un pulsante
button	N/A	Un pulsante generico
reset	N/A	Un pulsante la cui funzione è quella di resettare tutti gli altri campi ai loro valori iniziali

L'aspetto dei tipi di input `password`, `search`, `tel`, `url` e `email` non differiscono dal tipo standard `text`. Il loro scopo è quello di offrire suggerimenti al browser circa il contenuto previsto per quel campo di input, così il browser o lo script in esecuzione sul lato client possono intraprendere azioni personalizzate per uno specifico tipo di input. L'unica differenza tra il tipo di input testuale e il tipo di campo `password`, per esempio, è che il contenuto del campo `password` non viene visualizzato mentre il visitatore lo digita. Nei dispositivi touch screen, dove il testo viene digitato con una tastiera su schermo, il browser può far apparire solo la tastiera numerica quando un input di tipo `tel` ottiene il focus. Un'altra azione possibile è quella di suggerire una lista di indirizzi `email` conosciuti quando un input di tipo `email` viene mostrato.

Anche il tipo `number` appare come una semplice immissione di testo, ma con frecce di incremento/decremento al suo fianco. Il suo utilizzo farà sì che la tastiera numerica appaia nei dispositivi touchscreen quando ha il focus.

Gli altri elementi di input hanno il proprio aspetto e comportamento. Il tipo `date`, per esempio, è reso secondo le impostazioni locali del formato della data e viene visualizzato un calendario quando ottiene il focus:

```
<form>  
  
<p>  
  <label for="date">Date:</label>  
  <input type="date" name="date" id="date">  
</p>  
  
</form>
```

Figure 24 mostra come la versione desktop di Firefox visualizza attualmente questo campo.

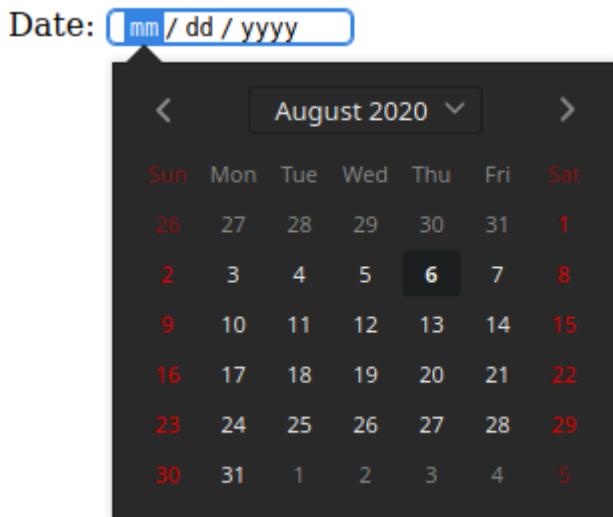


Figure 24. Il tipo di input della data.

NOTE

Gli elementi possono apparire leggermente diversi nei diversi browser o sistemi operativi, ma il loro funzionamento e utilizzo sono sempre gli stessi.

Questa è una caratteristica standard in tutti i browser moderni e non richiede opzioni extra o programmazione.

Indipendentemente dal tipo di input, il contenuto di un campo di input è chiamato il suo *valore*. Tutti i valori dei campi sono vuoti per impostazione predefinita, ma l'attributo `value` può essere usato per impostare un valore predefinito per il campo. Il valore per il tipo data deve usare il formato `YYYY-MM-DD`. Il valore predefinito nel seguente campo data è 6 agosto 2020:

```
<form>
  <p>
    <label for="date">Date:</label>
    <input type="date" name="date" id="date" value="2020-08-06">
  </p>
</form>
```

I tipi di input specifici aiutano il visitatore a riempire i campi, ma non gli impediscono di aggirare le restrizioni e inserire valori arbitrari in qualsiasi campo. Ecco perché è importante che i valori dei campi siano convalidati quando arrivano al server.

Gli elementi del modulo i cui valori devono essere digitati dal visitatore possono avere attributi speciali che aiutano a riempirli. L'attributo `placeholder` inserisce un valore di esempio nell'elemento di input:

```
<p>Address: <input type="text" name="address" id="address" placeholder="e.g. 41  
John St., Upper Suite 1"></p>
```

Il suggerimento appare all'interno dell'elemento di input, come mostrato in [Figure 25](#).

Address: e.g. 41 John St., Upper Suite 1

Figure 25. Esempio dell'attributo placeholder

Una volta che il visitatore inizia a digitare nel campo, il testo suggerito scompare. Il testo suggerito non viene inviato come valore del campo se il visitatore lascia il campo vuoto.

L'attributo `required` richiede che il visitatore compili un valore per il campo corrispondente prima di inviare il modulo:

```
<p>Address: <input type="text" name="address" id="address" required  
placeholder="e.g. 41 John St., Upper Suite 1"></p>
```

L'attributo `required` è un attributo *booleano*, quindi può essere messo da solo (senza il segno uguale). È importante contrassegnare i campi che sono richiesti, altrimenti il visitatore non sarà in grado di dire quali campi mancano e impediscono l'invio del modulo.

L'attributo `autocomplete` indica se il valore dell'elemento di `input` può essere completato automaticamente dal browser. Se impostato su `autocomplete="off"`, allora il browser non suggerisce valori precedenti per riempire il campo. Gli elementi di `input` per informazioni sensibili, come i numeri delle carte di credito, dovrebbero avere l'attributo `autocomplete` sempre impostato su `off`.

Inserimento di grandi quantità di testo: `textarea`

A differenza del campo di testo, dove si può inserire solo una riga di testo, l'elemento `textarea` permette al visitatore di inserire più di una riga di testo. La `textarea` è un elemento separato, ma non si basa sull'elemento `input`:

```
<p> <label for="comment">Type your comment here:</label> <br>
<textarea id="comment" name="comment" rows="10" cols="50">
My multi-line, plain-text comment.
</textarea>
</p>
```

L'aspetto tipico di una `textarea` è [Figure 26](#).

Type your comment here:

My multi-line, plain-text comment.

Figure 26. L'elemento `textarea`

Un'altra differenza rispetto all'elemento `input` è che l'elemento `textarea` ha un tag di chiusura (`</textarea>`), quindi il suo contenuto (cioè il suo valore) va in mezzo. Gli attributi `rows` e `cols` non

limitano la quantità di testo; sono usati solo per definire il *layout*. La textarea ha anche una maniglia nell'angolo in basso a destra, che permette al visitatore di ridimensionarla.

Liste di Opzioni

Si possono usare diversi tipi di controlli di modulo per presentare una lista di opzioni al visitatore: l'elemento `<select>` e i tipi di input `radio` e `checkbox`.

L'elemento `<select>` è un controllo a discesa con un elenco di voci predefinite:

```
<p><label for="browser">Favorite Browser:</label>
<select name="browser" id="browser">
<option value="firefox">Mozilla Firefox</option>
<option value="chrome">Google Chrome</option>
<option value="opera">Opera</option>
<option value="edge">Microsoft Edge</option>
</select>
</p>
```

Il tag `<option>` rappresenta una singola voce nel corrispondente controllo `<select>`. L'intero elenco viene visualizzato quando il visitatore tocca o fa clic sul controllo, come mostrato in [Figure 27](#).

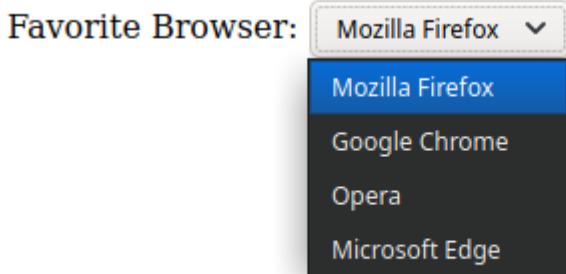


Figure 27. L'elemento `select`.

La prima voce della lista è selezionata per default. Per cambiare questo comportamento, puoi aggiungere l'attributo `selected` a un'altra voce in modo che sia selezionata al caricamento della pagina.

Il tipo di input `radio` è simile al controllo `<select>`, ma invece di un elenco a discesa, mostra tutte le voci in modo che il visitatore possa contrassegnarne una. I risultati del seguente codice sono mostrati in [Figure 28](#).

```

<p>Favorite Browser:</p>

<p>
  <input type="radio" id="browser-firefox" name="browser" value="firefox"
checked>
  <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
  <input type="radio" id="browser-chrome" name="browser" value="chrome">
  <label for="browser-chrome">Google Chrome</label>
</p>

<p>
  <input type="radio" id="browser-opera" name="browser" value="opera">
  <label for="browser-opera">Opera</label>
</p>

<p>
  <input type="radio" id="browser-edge" name="browser" value="edge">
  <label for="browser-edge">Microsoft Edge</label>
</p>

```

Favorite Browser:

- Mozilla Firefox
- Google Chrome
- Opera
- Microsoft Edge

Figure 28. Elementi di input di tipo radio.

Nota che tutti i tipi di input `radio` nello stesso gruppo hanno lo stesso attributo `name`. Ognuno di essi è esclusivo, quindi l'attributo `value` corrispondente per la voce scelta sarà quello associato all'attributo `name` condiviso. L'attributo `checked` funziona come l'attributo `selected` del controllo `<select>`. Contrassegna la voce corrispondente quando la pagina viene caricata per la prima volta. Il tag `<label>` è particolarmente utile per le voci radio, perché permette al visitatore di controllare una voce cliccando o toccando il testo corrispondente oltre al controllo stesso.

Mentre i controlli `radio` sono destinati a selezionare solo una singola voce di una lista, il tipo di input `checkbox` permette al visitatore di selezionare più voci:

```
<p>Favorite Browser:</p>

<p>
  <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
  <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
  <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
  <label for="browser-chrome">Google Chrome</label>
</p>

<p>
  <input type="checkbox" id="browser-opera" name="browser" value="opera">
  <label for="browser-opera">Opera</label>
</p>

<p>
  <input type="checkbox" id="browser-edge" name="browser" value="edge">
  <label for="browser-edge">Microsoft Edge</label>
</p>
```

Anche le caselle `checkbox` possono usare l'attributo `checked` per rendere le voci selezionate per default. Invece dei controlli rotondi dell'input `radio`, i `checkbox` sono resi come controlli quadrati, come mostrato in [Figure 29](#).

Favorite Browser:

- Mozilla Firefox
- Google Chrome
- Opera
- Microsoft Edge

Figure 29. Il tipo di input checkbox.

Se viene selezionata più di una voce, il browser le invierà con lo stesso nome, richiedendo allo sviluppatore di backend di scrivere codice specifico per leggere correttamente i dati del modulo che contiene checkbox.

Per migliorare l'usabilità, i campi di input possono essere raggruppati in un tag `<fieldset>`:

```
<fieldset>
<legend>Favorite Browser</legend>

<p>
    <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
        <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
    <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
        <label for="browser-chrome">Google Chrome</label>
</p>

<p>
    <input type="checkbox" id="browser-opera" name="browser" value="opera">
        <label for="browser-opera">Opera</label>
</p>

<p>
    <input type="checkbox" id="browser-edge" name="browser" value="edge">
        <label for="browser-edge">Microsoft Edge</label>
</p>
</fieldset>
```

Il tag `<legend>` contiene il testo che viene posizionato in cima alla cornice che il tag `<fieldset>` disegna intorno ai controlli ([Figure 30](#)).

Favorite Browser

- Mozilla Firefox
- Google Chrome
- Opera
- Microsoft Edge

Figure 30. Raggruppare gli elementi con il tag fieldset.

Il tag `<fieldset>` non cambia il modo in cui i valori dei campi sono inviati al server, ma permette allo sviluppatore frontend di controllare più facilmente i controlli annidati. Per esempio, impostando l'attributo `disabled` in un attributo `<fieldset>` si rendono tutti i suoi elementi interni non disponibili al visitatore.

L'Elemento `hidden`

Ci sono situazioni in cui lo sviluppatore vuole includere informazioni nel modulo che non possono essere manipolate dal visitatore. Per inviare un valore scelto dallo sviluppatore senza presentare un campo del modulo in cui il visitatore può digitare o cambiare il valore, lo sviluppatore potrebbe voler, per esempio, includere un *token* di identificazione per quel particolare modulo che non deve essere visto dal visitatore. Un elemento di modulo nascosto è codificato come nell'esempio seguente:

```
<input type="hidden" id="form-token" name="form-token" value="e730a375-b953-4393-847d-2dab065bbc92">
```

Il valore di un campo di input nascosto viene solitamente aggiunto al documento sul lato server, quando si restituisce al client il documento. Gli input nascosti sono trattati come campi ordinari quando il browser li invia al server.

Il Tipo di Input `File`

Oltre ai dati testuali, digitati o selezionati da una lista, i moduli HTML possono anche inviare file al server. Il tipo di input `file` permette al visitatore di scegliere un file dal file system locale e inviarlo direttamente dalla pagina web:

```
<p>
<label for="attachment">Attachment:</label><br>
<input type="file" id="attachment" name="attachment">
</p>
```

Invece di un campo del modulo in cui scrivere o da cui selezionare un valore, il tipo di input `file` mostra un pulsante `browse` che aprirà una finestra di dialogo del file. Qualsiasi tipo `file` è accettato dal tipo di input `file`, ma lo sviluppatore del backend probabilmente limiterà i tipi di file consentiti e la loro dimensione massima. La verifica del tipo di file può essere eseguita anche nel frontend aggiungendo l'attributo `accept`. Per accettare solo immagini JPEG e PNG, per esempio, l'attributo `accept` dovrebbe essere: `accept="image/jpeg, image/png"`.

Bottoni di Azione

Per impostazione predefinita, il modulo viene inviato quando il visitatore preme il tasto Enter in qualsiasi campo di input. Per rendere le cose più intuitive, si dovrebbe aggiungere un pulsante di invio con il tipo di input `submit`:

```
<input type="submit" value="Submit form">
```

Il testo nell'attributo `value` viene visualizzato sul pulsante, come mostrato in [Figure 31](#).

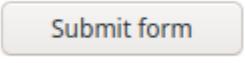


Figure 31. Un bottone standard `submit`.

Un altro pulsante utile da includere nei moduli complessi è il pulsante `reset`, che cancella il modulo e lo riporta al suo stato originale:

```
<input type="reset" value="Clear form">
```

Come il pulsante di invio, il testo nell'attributo `value` è usato per etichettare il pulsante. In alternativa, il tag `<button>` può essere usato per aggiungere pulsanti ai moduli o in qualsiasi altro punto della pagina. A differenza dei pulsanti realizzati con il tag `<input>`, l'elemento `button` ha un tag di chiusura e l'etichetta è il suo contenuto interno:

```
<button>Submit form</button>
```

All'interno di un modulo, l'azione predefinita per l'elemento `button` è di inviare il modulo. Come per i pulsanti di `input`, l'attributo `type` del pulsante può essere cambiato in `reset`.

Azione e Metodi nei Moduli

L'ultimo passo nella scrittura di un modulo HTML è definire come e dove i dati devono essere inviati. Questi aspetti dipendono da dettagli sia nel client sia nel server.

Lato server, l'approccio più comune è quello di avere un file di script che analizza, convalida ed elabora i dati del modulo secondo lo scopo dell'applicazione. Per esempio, lo sviluppatore di backend potrebbe scrivere uno script chiamato `receive_form.php` per ricevere i dati inviati dal modulo. Sul lato client, lo script è indicato nell'attributo `action` del tag del modulo:

```
<form action="receive_form.php">
```

L'attributo `action` segue le stesse convenzioni di tutti gli indirizzi HTTP. Se lo script si trova nello stesso livello gerarchico della pagina che contiene il modulo, può essere scritto senza il suo percorso. Altrimenti, il percorso assoluto o relativo deve essere fornito. Lo script dovrebbe anche generare la risposta per servire come pagina di destinazione, caricata dal browser dopo che il visitatore ha inviato il modulo.

HTTP fornisce metodi distinti per inviare i dati di un modulo attraverso una connessione con il server. I metodi più comuni sono `get` e `post`, che dovrebbero essere indicati nell'attributo `method` del tag `form`:

```
<form action="receive_form.php" method="get">
```

Ottimamente:

```
<form action="receive_form.php" method="post">
```

Nel metodo `get`, i dati del modulo sono codificati direttamente nell'URL della richiesta. Quando il visitatore invia il modulo, il browser caricherà l'URL definito nell'attributo `action` con i campi del modulo aggiunti.

Il metodo `get` è preferito per piccole quantità di dati, come i semplici moduli di contatto. Tuttavia, l'URL non può superare alcune migliaia di caratteri, quindi il metodo `post` dovrebbe essere usato quando i moduli contengono campi grandi o non testuali, come le immagini.

Il metodo `post` fa sì che il browser invii i dati del modulo nella sezione del corpo della richiesta HTTP. Sebbene sia necessario per i dati binari che superano il limite di dimensione di un URL, il metodo `post` aggiunge un inutile sovraccarico alla connessione quando viene usato in moduli testuali più semplici, quindi il metodo `get` è preferito in questi casi.

Il metodo scelto non influenza il modo in cui il visitatore interagisce con il modulo. I metodi `get` e `post` sono processati diversamente dallo script lato server che riceve il modulo.

Quando si usa il metodo `post`, è anche possibile cambiare il tipo MIME del contenuto del modulo con l'attributo del modulo `enctype`. Questo influisce su come i campi e i valori del modulo saranno impilati insieme nella comunicazione HTTP con il server. Il valore predefinito per `enctype` è `application/x-www-form-urlencoded`, che è simile al formato usato nel metodo `get`. Se il modulo contiene campi di input di tipo `file`, dovrebbe invece essere usato l'`enctype multipart/form-data`.

Esercizi Guidati

1. Qual è il modo corretto di associare un tag <label> a un tag <input>?

2. Quale tipo di elemento di input fornisce un controllo a scorrimento per scegliere un valore numerico?

3. Qual è lo scopo dell'attributo di modulo placeholder?

4. Come si può rendere selezionata di default la seconda opzione in un controllo select?

Esercizi Esplorativi

1. Come si può cambiare un input di file per fargli accettare solo file PDF?

2. Come si può informare il visitatore su quali campi di un modulo sono richiesti?

3. Metti insieme le tre parti di codice di questa lezione in un singolo modulo. Assicurati di non usare lo stesso attributo `name` o `id` in più controlli del modulo.

Sommario

Questa lezione tratta del come creare semplici moduli HTML per inviare dati al server. Sul lato client, i form HTML consistono in elementi HTML standard che sono combinati per costruire interfacce personalizzate. Inoltre, i form devono essere configurati per comunicare correttamente con il server. La lezione passa attraverso i seguenti concetti e procedure:

- Il tag `<form>` e la struttura base del modulo.
- Elementi di input di base e speciali.
- Il ruolo dei tag speciali come `<label>`, `<fieldset>` e `<legend>`.
- I pulsanti e le azioni del modulo.

Risposte agli Esercizi Guidati

1. Qual è il modo corretto di associare un tag `<label>` a un tag `<input>`?

L'attributo `for` del tag `<label>` dovrebbe contenere l'id del corrispondente tag `<input>`.

2. Quale tipo di elemento di input fornisce un controllo a scorrimento per scegliere un valore numerico?

L'elemento di input `range`.

3. Qual è lo scopo dell'attributo di modulo `placeholder`?

L'attributo `placeholder` contiene un esempio di input che è visibile quando il campo di input corrispondente è vuoto.

4. Come si può rendere selezionata di default la seconda opzione in un controllo select?

Il secondo elemento `option` dovrebbe avere l'attributo `selected`.

Risposte agli Esercizi Esplorativi

1. Come si può cambiare un input di file per fargli accettare solo file PDF?

L'attributo `accept` dell'elemento di input dovrebbe essere impostato su `application/pdf`.

2. Come si può informare il visitatore su quali campi di un modulo sono richiesti?

Di solito, i campi obbligatori sono marcati con un asterisco (*), e una breve nota come “I campi marcati con * sono obbligatori” è posta vicino al modulo.

3. Metti insieme le tre parti di codice di questa lezione in un singolo modulo. Assicurati di non usare lo stesso attributo `name` o `id` in più controlli del modulo.

```
<form action="receive_form.php" method="get">

<h2>Personal Information</h2>

<label for="fullname">Full name:</label>
<p><input type="text" name="fullname" id="fullname"></p>

<p>
    <label for="date">Date:</label>
    <input type="date" name="date" id="date">
</p>

<p>Favorite Browser:</p>

<p>
    <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
    <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
    <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
    <label for="browser-chrome">Google Chrome</label>
</p>

<p>
    <input type="checkbox" id="browser-opera" name="browser" value="opera">
    <label for="browser-opera">Opera</label>
</p>

<p>
    <input type="checkbox" id="browser-edge" name="browser" value="edge">
    <label for="browser-edge">Microsoft Edge</label>
</p>

<p><input type="reset" value="Clear form"></p>
<p><input type="submit" value="Submit form"></p>

</form>
```



Argomento 033: Stile dei Contenuti con i CSS



033.1 Concetti Base dei CSS

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 033.1

Peso

1

Arese di Conoscenza Chiave

- Incorporare i CSS in un documento HTML
- Comprendere la sintassi CSS
- Aggiungere commenti ai CSS
- Conoscenza delle caratteristiche e dei requisiti di accessibilità

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- Attributi HTML `style` e `type` (text/css)
- `<style>`
- `<link>`, inclusi gli attributi `rel` (stylesheet), `type` (text/css) e `src`
- ;
- / , /



033.1 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	033 Stile dei Contenuti con i CSS
Obiettivo:	033.1 Concetti Base dei CSS
Lezione:	1 di 1

Introduzione

Tutti i browser web mostrano le pagine HTML usando regole di presentazione predefinite che sono pratiche e dirette, ma non visivamente attraenti. L'HTML di per sé offre poche caratteristiche per scrivere pagine elaborate basate sui moderni concetti di *User Experience*. Dopo aver scritto semplici pagine HTML, ti sarai probabilmente reso conto che sono antiestetiche se paragonate a pagine ben progettate trovi su Internet. Questo perché, nel moderno HTML, il codice di *markup* destinato alla struttura e alla funzione degli elementi nel documento (cioè, il *contenuto semantico*) è separato dalle regole che definiscono come gli elementi dovrebbero apparire (la *presentazione*). Le regole di presentazione sono scritte in un linguaggio diverso chiamato *Cascading Style Sheets* (CSS). Esso permette di cambiare quasi tutti gli aspetti visivi del documento, come i caratteri, i colori, e il posizionamento degli elementi lungo la pagina. I documenti HTML non sono, nella maggior parte dei casi, destinati a essere visualizzati in un layout fisso come un file PDF. Piuttosto, le pagine web basate su HTML saranno probabilmente visualizzate su un'ampia varietà di dimensioni dello schermo o anche stampate. I CSS forniscono opzioni regolabili per assicurare che la pagina web sia resa come previsto, adattata al dispositivo o all'applicazione che la apre.

Applicare gli Stili

Ci sono diversi modi per includere i CSS in un documento HTML: scriverli direttamente nel tag dell'elemento, in una sezione separata del codice sorgente della pagina, o in un file esterno a cui fare riferimento nella pagina HTML.

L'Attributo `style`

Il modo più semplice per modificare lo stile di un elemento specifico è quello di scriverlo direttamente nel tag dell'elemento usando l'attributo `style`. Tutti gli elementi HTML visibili permettono un attributo `style`, il cui valore può essere una o più regole CSS, note anche come *proprietà*. Iniziamo con un semplice esempio, un elemento paragrafo:

```
<p>My stylized paragraph</p>
```

La sintassi di base di una proprietà CSS personalizzata è `property: value`, dove `property` è il particolare aspetto che vuoi cambiare nell'elemento e `value` definisce la sostituzione dell'opzione di default fatta dal browser. Alcune proprietà si applicano a *tutti* gli elementi e alcune proprietà si applicano *solo* a elementi specifici. Allo stesso modo, ci sono valori appropriati da usare in ogni proprietà.

Per cambiare il colore del nostro semplice paragrafo, per esempio, usiamo la proprietà `color`. La proprietà `color` si riferisce al colore *foreground*, cioè il colore delle lettere nel paragrafo. Il colore stesso va nella parte del valore della regola e può essere specificato in molti formati diversi, inclusi nomi semplici come `red`, `green`, `blue`, `yellow` e così via. Quindi, per rendere la lettera del paragrafo `purple`, aggiungi la proprietà personalizzata `color: purple` all'attributo `style` dell'elemento:

```
<p style="color: purple">My first stylized paragraph</p>
```

Altre proprietà personalizzate possono andare nella stessa proprietà `style`, ma devono essere separate da punto e virgola. Se vuoi rendere la dimensione del carattere più grande, per esempio, aggiungi `font-size: larger` alla proprietà `style`:

```
<p style="color: purple; font-size: larger">My first stylized paragraph</p>
```

NOTE

Non è necessario aggiungere spazi intorno ai due punti e ai punti e virgola, ma possono rendere più facile la lettura del codice CSS.

Per vedere il risultato di queste modifiche, salvate il seguente HTML in un file e poi apritelo in un browser web (i risultati sono mostrati in [Figure 32](#)):

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>
</head>
<body>

<p style="color: purple; font-size: larger">My first stylized paragraph</p>

<p style="color: green; font-size: larger">My second stylized paragraph</p>

</body>
</html>
```

My first stylized paragraph

My second stylized paragraph

Figure 32. Un cambiamento visivo molto semplice usando i CSS.

Puoi immaginare ogni elemento della pagina come un rettangolo o una scatola cui puoi modificare, tramite i CSS, proprietà geometriche e decorazioni. Il meccanismo di rendering usa due concetti standard di base per il posizionamento degli elementi: il posizionamento a *blocco* e il posizionamento *in linea*. Gli elementi a blocco occupano tutto lo spazio orizzontale del loro elemento padre e sono posizionati in modo sequenziale, dall'alto verso il basso. La loro altezza (la loro *dimensione verticale*, da non confondere con la loro posizione *in alto* nella pagina) dipende generalmente dalla quantità di contenuto che hanno. Gli elementi in linea seguono il metodo da sinistra a destra simile alle lingue scritte occidentali: gli elementi sono posizionati orizzontalmente, da sinistra a destra, fino a quando non c'è più spazio sul lato destro, dopodiché l'elemento continua su una nuova linea proprio sotto quella corrente. Elementi come p, div e section sono posizionati di default come blocchi, mentre elementi come span, em, a e singole lettere sono posizionati in linea. Questi metodi di posizionamento di base possono essere fondamentalmente modificati dalle regole CSS.

Il rettangolo corrispondente all'elemento p nel corpo del documento HTML di esempio sarà visibile se si aggiunge la proprietà background-color alla regola ([Figure 33](#)):

```
<p style="color: purple; font-size: larger; background-color: silver">My first  
stylized paragraph</p>
```

```
<p style="color: green; font-size: larger; background-color: silver">My second  
stylized paragraph</p>
```

My first stylized paragraph

My second stylized paragraph

Figure 33. Rettangoli corrispondenti ai paragrafi.

Man mano che aggiungi nuove proprietà CSS personalizzate all’attributo `style`, noterai che il codice complessivo inizia a diventare disordinato. Scrivere troppe regole CSS nell’attributo `style` mina la separazione tra struttura (HTML) e presentazione (CSS). Inoltre, ti renderai presto conto che molti stili sono condivisi tra diversi elementi e non è saggio ripeterli in ogni elemento.

Regole dei CSS

Invece di applicare lo stile agli elementi direttamente nei loro attributi `style`, è molto più pratico indicare al browser l’intera collezione di elementi a cui si applica lo stile personalizzato. Lo facciamo aggiungendo un *selettore* alle proprietà personalizzate, abbinando elementi per tipo, classe, ID unico, posizione relativa e così via. La combinazione di un *selettore* e delle corrispondenti proprietà personalizzate—conosciuta anche come *dichiarazione*—è chiamata *regola CSS*. La sintassi di base di una regola CSS è `selector { property: value }`. Come nell’attributo `style`, le proprietà separate da punti e virgola possono essere raggruppate insieme, come in `p { color: purple; font-size: larger }`. Questa regola corrisponde a ogni elemento `p` nella pagina e applica le proprietà personalizzate `color` e `font-size`.

Una regola CSS di un elemento genitore corrisponderà automaticamente a tutti i suoi elementi figli. Questo significa che potremmo applicare le proprietà personalizzate a tutto il testo nella pagina, indipendentemente dal fatto che sia dentro o fuori un tag `<p>`, usando invece il selettore `body: body { color: purple; font-size: larger }`. Questa strategia ci libera dallo scrivere di nuovo la stessa regola per tutti i figli, ma potrebbe essere necessario scrivere regole aggiuntive per “annullare” o modificare le regole ereditate.

Il Tag `style`

Il tag `<style>` ci permette di scrivere regole CSS all’interno della pagina HTML che vogliamo “stilizzare”. Permette al browser di differenziare il codice CSS dal codice HTML. Il tag `<style>` va

nella sezione head del documento:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>

  <style type="text/css">
    p { color: purple; font-size: larger }
  </style>

</head>
<body>

<p>My first stylized paragraph</p>

<p>My second stylized paragraph</p>

</body>
</html>
```

L'attributo `type` dice al browser che tipo di contenuto è dentro il tag `<style>`, cioè il suo *MIME type*. Poiché ogni browser che supporta i CSS assume che il tipo del tag `<style>` sia di default `text/css`, includere l'attributo `type` è opzionale. C'è anche un attributo `media`—schermi di computer o stampanti, per esempio—a cui si applicano le regole CSS nel tag `<style>`. Per impostazione predefinita, le regole CSS si applicano a *qualsiasi supporto* in cui il documento venga visualizzato.

Come nel codice HTML, le interruzioni di riga e l'indentazione del codice non cambiano il modo in cui le regole CSS sono interpretate dal browser. Quindi scrivere:

```
<style type="text/css">
  p { color: purple; font-size: larger }
</style>
```

ha lo stesso risultato di scrivere:

```
<style type="text/css">  
  
p {  
    color: purple;  
    font-size: larger;  
}  
  
</style>
```

I *byte extra* utilizzati dagli spazi e dalle interruzioni di riga fanno poca differenza nella dimensione finale del documento e non hanno un impatto significativo sul tempo di caricamento della pagina, quindi la scelta del layout è una questione di gusto. Nota il punto e virgola dopo l'ultima dichiarazione (`font-size: larger;`) della regola CSS. Quel punto e virgola non è obbligatorio, ma averlo li rende più facile aggiungere un'altra dichiarazione nella riga successiva senza preoccuparsi dei punti e virgola mancanti.

Avere le dichiarazioni in linee separate aiuta anche quando hai bisogno di commentare una dichiarazione. Ogni volta che vuoi disabilitare temporaneamente una dichiarazione per ragioni di risoluzione dei problemi, per esempio, puoi racchiudere la linea corrispondente con `/*` e `*/`:

```
p {  
    color: purple;  
/*  
    font-size: larger;  
*/  
}
```

o:

```
p {  
    color: purple;  
    /* font-size: larger; */  
}
```

Scritta così, la dichiarazione `font-size: larger` sarà ignorata dal browser. Fai attenzione ad aprire e chiudere correttamente i commenti, altrimenti il browser potrebbe non essere in grado di interpretare le regole.

I commenti sono anche utili per scrivere promemoria sulle regole:

```
/* Texts inside <p> are purple and larger */
p {
  color: purple;
  font-size: larger;
}
```

I promemoria come quello nell'esempio sono sacrificabili nei fogli di stile che contengono un piccolo numero di regole, ma sono essenziali per aiutare la navigazione dei fogli di stile con centinaia (o più) di regole.

Anche se l'attributo `style` e il tag `<style>` sono adeguati per testare stili personalizzati e utili per situazioni specifiche, non vengono usati comunemente. Invece, le regole CSS sono solitamente tenute in un file separato che può essere referenziato dal documento HTML.

I CSS in File Esterne

Il metodo preferito per associare le definizioni CSS a un documento HTML è quello di memorizzare i CSS in un file separato. Questo metodo offre due vantaggi principali rispetto ai precedenti:

- Le stesse regole di stile possono essere condivise tra documenti diversi.
- Il file CSS viene solitamente memorizzato nella cache dal browser, migliorando i tempi di caricamento futuri.

I file CSS hanno l'estensione `.css` e, come i file HTML, possono essere modificati con qualsiasi editor di testo semplice. A differenza dei file HTML, i file CSS non hanno una struttura di livello superiore costruita con tag gerarchici come `<head>` o `<body>`. Piuttosto, il file CSS è solo una lista di regole processate in ordine sequenziale dal browser. Le stesse regole scritte dentro un tag `<style>` potrebbero invece andare in un file CSS.

L'associazione tra il documento HTML e le regole CSS memorizzate in un file è definita solo nel documento HTML. Al file CSS non importa se esistono elementi che corrispondono alle sue regole, quindi non c'è bisogno di enumerare nel file CSS i documenti HTML a cui è collegato. Sul lato HTML, ogni foglio di stile collegato verrà applicato al documento, proprio come se le regole fossero scritte in un tag `<style>`.

Il tag HTML `<link>` definisce un foglio di stile esterno da usare nel documento corrente e dovrebbe andare nella sezione `head` del documento HTML:

```
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>

  <link href="style.css" rel="stylesheet">

</head>
```

Ora puoi mettere la regola per l'elemento `p` che abbiamo usato prima nel file `style.css`, e i risultati visti dal visitatore della pagina web saranno gli stessi. Se il file CSS non è nella stessa directory del documento HTML, specifica il suo percorso relativo o assoluto nell'attributo `href`. Il tag `<link>` può riferirsi a diversi tipi di risorse esterne, quindi è importante stabilire quale relazione ha la risorsa esterna con il documento corrente. Per i file CSS esterni, la relazione è definita da `rel="stylesheet"`.

L'attributo `media` può essere usato nello stesso modo del tag `<style>`: indica i media, come gli schermi del computer o la stampa, ai quali si devono applicare le regole del file esterno.

I CSS possono cambiare completamente un elemento, ma è comunque importante usare l'elemento appropriato per i componenti della pagina. Questo dovrebbe evitare che le tecnologie assistive come gli *screen-reader* possano non essere in grado di identificare le sezioni corrette della pagina.

Esercizi Guidati

1. Quali metodi si possono usare per cambiare l'aspetto degli elementi HTML usando i CSS?

2. Perché non si raccomanda di usare l'attributo `style` del tag `<p>` se ci sono paragrafi fratelli che devono avere lo stesso aspetto?

3. Qual è la politica di posizionamento predefinita nel posizionare un elemento `div`?

4. Quale attributo del tag `<link>` indica la posizione di un file CSS esterno?

5. Qual è la sezione corretta per inserire l'elemento `link` in un documento HTML?

Esercizi Esplorativi

1. Perché non si raccomanda di usare un tag `<div>` per identificare una parola in una frase ordinaria?

2. Che cosa succede se inizi un commento con `/*` nel mezzo di un file CSS, ma dimentichi di chiuderlo con `*/`?

3. Scrivi una regola CSS per disegnare una sottolineatura in tutti gli elementi `em` del documento.

4. Come si può indicare da un tag `<style>` o `<link>` che un foglio di stile dovrebbe essere usato solo dai sintetizzatori vocali?

Sommario

Questa lezione tratta i concetti di base dei CSS e come integrarli con l'HTML. Le regole scritte nei fogli di stile CSS sono il metodo standard per cambiare l'aspetto dei documenti HTML. I CSS ci permettono di mantenere il contenuto semantico separato dalle politiche di presentazione. Questa lezione passa attraverso i seguenti concetti e procedure:

- Come cambiare le proprietà CSS usando l'attributo `style`.
- La sintassi di base delle regole CSS.
- Usare il tag `<style>` per incorporare le regole CSS nel documento.
- Usare il tag `<link>` per incorporare fogli di stile esterni al documento.

Risposte agli Esercizi Guidati

1. Quali metodi si possono usare per cambiare l'aspetto degli elementi HTML usando i CSS?

Tre metodi di base: Scriverlo direttamente nel tag dell'elemento, in una sezione separata del codice sorgente della pagina; o in un file esterno cui fare riferimento nella pagina HTML.

2. Perché non si raccomanda di usare l'attributo `style` del tag `<p>` se ci sono paragrafi fratelli che devono avere lo stesso aspetto?

La dichiarazione CSS dovrà essere replicata negli altri tag `<p>`, il che richiede tempo, aumenta la dimensione del file ed è soggetto a errori.

3. Qual è la politica di posizionamento predefinita nel posizionare un elemento `div`?

L'elemento `div` è trattato come un elemento di blocco per default, quindi occuperà tutto lo spazio orizzontale del suo elemento genitore e l'altezza dipenderà dal suo contenuto.

4. Quale attributo del tag `<link>` indica la posizione di un file CSS esterno?

L'attributo `href`.

5. Qual è la sezione corretta per inserire l'elemento `link` in un documento HTML?

L'elemento `link` dovrebbe essere nella sezione `head` del documento HTML..

Risposte agli Esercizi Esplorativi

1. Perché non si raccomanda di usare un tag `<div>` per identificare una parola in una frase ordinaria?

Il tag `<div>` fornisce una separazione semantica tra due sezioni distinte del documento e interferisce con gli strumenti di accessibilità quando è usato per elementi di testo in linea.

2. Che cosa succede se inizi un commento con `/*` nel mezzo di un file CSS, ma dimentichi di chiuderlo con `*/`?

Tutte le regole dopo il commento saranno ignorate dal browser.

3. Scrivi una regola CSS per disegnare una sottolineatura in tutti gli elementi `em` del documento.

```
em { text-decoration: underline }
```

o

```
em { text-decoration-line: underline }
```

4. Come si può indicare da un tag `<style>` o `<link>` che un foglio di stile dovrebbe essere usato solo dai sintetizzatori vocali?

Il valore dell'attributo `media` deve essere `speech`.



033.2 Selettori CSS e Applicazione di Stili

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 033.2

Peso

3

Arese di Conoscenza Chiave

- Utilizzare i selettori per applicare le regole CSS agli elementi
- Comprendere le pseudo-classi CSS
- Comprendere l'ordine delle regole e la precedenza nei CSS
- Comprendere l'ereditarietà nei CSS

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- element; .class; #id
- a, b; a.class; a b;
- :hover, :focus
- !important



033.2 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	033 Stile dei Contenuti con i CSS
Obiettivo:	033.2 Selettori CSS e Applicazione di Stili
Lezione:	1 di 1

Introduzione

Quando scriviamo una regola CSS, dobbiamo indicare al browser a quali elementi si applica la regola. Lo facciamo specificando un *selettore*: uno schema che può corrispondere a un elemento o a un gruppo di elementi. I selettori sono disponibili in molte forme diverse, basate sul nome dell'elemento, i suoi attributi, il suo posizionamento relativo nella struttura del documento, o una combinazione di queste caratteristiche.

Stili per l'Intera Pagina

Uno dei vantaggi di usare i CSS è che *non è necessario* scrivere regole individuali per gli elementi che condividono lo stesso stile. Un asterisco applica la regola a tutti gli elementi della pagina web, come mostrato nel seguente esempio:

```
* {  
    color: purple;  
    font-size: large;  
}
```

Il selettor * non è l'unico metodo per applicare una regola di stile a tutti gli elementi della pagina. Un selettor che semplicemente corrisponde a un elemento per il suo nome di tag è chiamato un *selettor di tipo*, quindi qualsiasi nome di tag HTML come body, p, table, em e così via può essere usato come selettor. Nei CSS, lo stile è *ereditato* dai suoi elementi figli. Quindi, in pratica, usare il selettor * ha lo stesso effetto di applicare una regola all'elemento body:

```
body {  
    color: purple;  
    font-size: large;  
}
```

Inoltre, la caratteristica “a cascata” dei CSS permette di mettere a punto le proprietà ereditate di un elemento. Puoi scrivere una regola CSS generale che si applica a tutti gli elementi della pagina, e poi scrivere regole per elementi specifici o gruppi di elementi.

Se lo stesso elemento corrisponde a due o più regole in conflitto, il browser applica la regola del selettor più specifico. Prendiamo come esempio le seguenti regole CSS:

```
body {  
    color: purple;  
    font-size: large;  
}  
  
li {  
    font-size: small;  
}
```

Il browser applicherà gli stili color: purple e font-size: large a tutti gli elementi dentro l'elemento body. Tuttavia, se ci sono elementi li nella pagina, il browser sostituirà lo stile font-size: large con lo stile font-size: small, perché il selettor li ha una relazione più forte con quest'ultimo rispetto al selettor body.

I CSS non limitano il numero di selettori equivalenti nello stesso foglio di stile, quindi puoi avere due o più regole che usano lo stesso selettor:

```
li {
  font-size: small;
}
```

```
li {
  font-size: x-small;
}
```

In questo caso, entrambe le regole sono ugualmente specifiche per l'elemento `li`. Il browser non può applicare regole contrastanti, quindi sceglierà la regola che viene dopo nel file CSS. Per evitare confusione, la raccomandazione è di raggruppare tutte le proprietà che usano lo stesso selettori.

L'ordine in cui le regole appaiono nel foglio di stile influisce su come vengono applicate nel documento, ma puoi sovrascrivere questo comportamento usando una regola *importante*. Se, per qualsiasi ragione, vuoi mantenere le due regole `li` separate, ma forzare l'applicazione della prima invece della seconda, segna la prima regola come importante:

```
li {
  font-size: small !important;
}
```

```
li {
  font-size: x-small;
}
```

Le regole marcate con `!important` dovrebbero essere usate con cautela, perché rompono la naturale sequenza dei fogli di stile e rendono più difficile trovare e correggere i problemi all'interno del file CSS.

Selettori Restrittivi

Abbiamo visto come poter cambiare certe proprietà ereditate usando selettori che corrispondono a tag specifici. Tuttavia, di solito abbiamo bisogno di usare stili distinti per elementi dello stesso tipo.

Gli attributi dei tag HTML possono essere incorporati nei selettori per limitare l'insieme degli elementi a cui si riferiscono. Supponiamo che la pagina HTML su cui stai lavorando abbia due tipi di liste non ordinate (``): una è usata all'inizio della pagina come menu per le sezioni del sito web e l'altro tipo è usato per liste convenzionali nel corpo del testo:

```
<!DOCTYPE html>
```

```
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>

<div id="content">

<p>The three rocky planets of the solar system are:</p>

<ul>
  <li>Mercury</li>
  <li>Venus</li>
  <li>Earth</li>
  <li>Mars</li>
</ul>

<p>The outer giant planets made most of gas are:</p>

<ul>
  <li>Jupiter</li>
  <li>Saturn</li>
  <li>Uranus</li>
  <li>Neptune</li>
</ul>

</div><!-- #content -->

<div id="footer">

<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>
```

```
</div><!-- #footer -->
</body>
</html>
```

Per impostazione predefinita, ogni elemento della lista ha un pallino nero alla sua sinistra. Potresti voler rimuovere i pallini dall'elenco del menu in alto lasciandoli nell'altro elenco. Tuttavia, non puoi semplicemente usare il selettore `li` perché così facendo rimuoveresti anche i pallini nell'elenco all'interno della sezione del corpo del testo. Hai bisogno di un modo per dire al browser di modificare solo gli elementi dell'elenco usati in un elenco, ma non nell'altro.

Ci sono diversi modi per scrivere selettori che corrispondano a specifici elementi nella pagina. Come detto prima, vedremo prima come usare gli attributi degli elementi per farlo. Per questo esempio in particolare possiamo usare l'attributo `id` per specificare *solo* la lista superiore.

L'attributo `id` assegna un identificatore unico all'elemento corrispondente, che possiamo usare come selettore della regola CSS. Prima di scrivere la regola CSS, modifica il file HTML di esempio e aggiungi `id="topmenu"` all'elemento `ul` usato per il menu superiore:

```
<ul id="topmenu">
  <li>Home</li>
  <li>Articles</li>
  <li>About</li>
</ul>
```

C'è già un elemento `link` nella sezione `head` del documento HTML che punta al file del foglio di stile `style.css` nella stessa cartella, quindi possiamo aggiungervi le seguenti regole CSS:

```
ul#topmenu {
  list-style-type: none
}
```

Il carattere *hash* è usato in un selettore, dopo un elemento, per designare un ID (senza spazi di separazione). Il nome del tag a sinistra dell'hash è opzionale, poiché non ci saranno altri elementi con lo stesso ID. Pertanto, il selettore potrebbe essere scritto semplicemente come `#topmenu`.

Anche se la proprietà `list-style-type` *non* è una proprietà diretta dell'elemento `ul`, le proprietà CSS dell'elemento genitore sono ereditate dai suoi figli, quindi lo stile assegnato all'elemento `ul` sarà ereditato dai suoi elementi `li` figli.

Non tutti gli elementi hanno un ID attraverso il quale possono essere selezionati. Solo alcuni elementi chiave del layout di una pagina dovrebbero avere un ID. Considera le liste di pianeti usate precedentemente nel codice di esempio. Sebbene sia possibile assegnare ID unici per ogni singolo elemento ripetuto, questo metodo non è pratico per pagine più lunghe con molti contenuti. Piuttosto, possiamo usare l'ID dell'elemento padre `div` come selettore per cambiare le proprietà dei suoi elementi interni.

Tuttavia l'elemento `div` non è direttamente collegato alle liste HTML, quindi aggiungere la proprietà `list-style-type` a esso non avrà alcun effetto sui suoi figli. Quindi, per cambiare il pallino nero nelle liste all'interno del contenuto `div` in un pallino vuoto, dovremmo usare un selettore *descendente*:

```
#topmenu {  
    list-style-type: none  
}  
  
#content ul {  
    list-style-type: circle  
}
```

Il selettore `#content ul` è chiamato *selettore descendente* perché corrisponde solo agli elementi `ul` che sono figli dell'elemento il cui ID è `content`. Possiamo usare tanti livelli di discendenza quanti sono necessari. Per esempio, usando `#content ul li` si troverebbero solo gli elementi `li` che sono discendenti di elementi `ul` che sono discendenti dell'elemento il cui ID è `content`. Ma in questo esempio, il selettore più lungo avrà lo stesso effetto dell'uso di `#content ul`, perché gli elementi `li` ereditano le proprietà CSS impostate sul loro genitore `ul`. I selettori discendenti sono una tecnica essenziale quando il layout della pagina cresce in complessità.

Diciamo che ora vuoi cambiare la proprietà `font-style` degli elementi della lista nel `topmenu` e nella lista nel `footer div` per farli sembrare obliqui. Non puoi semplicemente scrivere una regola CSS usando `ul` come selettore, perché cambierebbe anche gli elementi della lista nel `content div`. Finora abbiamo cambiato le proprietà CSS usando un selettore alla volta, e questo metodo può essere usato anche per questo compito:

```
#topmenu {
    font-style: oblique
}

#footer ul {
    font-style: oblique
}
```

I selettori separati però non sono l'unico modo per farlo. I CSS ci permettono di raggruppare i selettori che condividono uno o più stili, usando una lista di selettori separati da virgole:

```
#topmenu, #footer ul {
    font-style: oblique
}
```

Raggruppare i selettori elimina il lavoro extra di scrivere stili duplicati. Inoltre, potresti voler cambiare nuovamente la proprietà in futuro e potresti non ricordarti tutti i diversi luoghi dove doverla cambiarla.

Classi

Se non vuoi preoccuparti troppo della gerarchia degli elementi, puoi semplicemente aggiungere una classe all'insieme di elementi che desideri personalizzare. Le classi sono simili agli ID, ma invece di identificare un solo elemento nella pagina, possono identificare un gruppo di elementi che condividono le stesse caratteristiche.

Prendi la pagina HTML di esempio su cui stiamo lavorando. È improbabile che nelle pagine del mondo reale troveremo strutture così semplici, quindi sarebbe più pratico selezionare un elemento utilizzando solo le classi o una combinazione di classi e relativa discendenza. Per applicare la proprietà `font-style: oblique` agli elenchi di menu utilizzando le classi, prima dobbiamo aggiungere la proprietà `class` agli elementi nel file HTML. Lo faremo prima nel menu in alto:

```
<ul id="topmenu" class="menu">
    <li><a href="/">Home</a></li>
    <li><a href="/articles">Articles</a></li>
    <li><a href="/about">About</a></li>
</ul>
```

E poi nel menu a piè di pagina:

```
<div id="footer">

<ul class="menu">
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>

</div><!-- #footer -->
```

Fatto ciò, possiamo sostituire il gruppo di selettori `#topmenu, #footer ul` con il selettore basato sulla classe `.menu`:

```
.menu {
  font-style: oblique
}
```

Come per i selettori basati su ID, l'aggiunta del nome del tag a sinistra del punto nei selettori basati su classi è opzionale. Tuttavia, a differenza degli ID, si suppone che la stessa classe possa essere usata in più di un elemento e non è nemmeno necessario che sia dello stesso tipo. Perciò, se la classe `menu` è condivisa tra diversi tipi di elementi, usare il selettore `ul.menu` corrisponderebbe solo agli elementi `ul` che hanno la classe `menu`. Invece, usando `.menu` come selettore si abbinerà a qualsiasi elemento che abbia la classe `menu`, indipendentemente dal suo tipo.

Inoltre, gli elementi possono essere associati a più di una classe. Potremmo differenziare il menu in alto e quello in basso aggiungendo una classe in più a ciascuno di essi:

```
<ul id="topmenu" class="menu top">
```

E nel menu a piè di pagina:

```
<ul class="menu footer">
```

Quando l'attributo `class` ha più di una classe, queste devono essere separate da spazi. Ora, oltre alla regola CSS condivisa tra gli elementi della classe `menu`, possiamo indirizzare il menu in alto e a piè di pagina usando le loro classi corrispondenti:

```
.menu {
    font-style: oblique
}

.menu.top {
    font-size: large
}

.menu.footer {
    font-size: small
}
```

Fai attenzione che scrivere `.menu.top` differisce da `.menu .top` (con uno spazio tra le parole). Il primo selettori corrisponderà agli elementi che hanno entrambe le classi `menu` e `top`, mentre il secondo corrisponderà agli elementi che hanno la classe `top` e un elemento padre con la classe `menu`.

Selettori Speciali

I selettori CSS possono anche corrispondere a stati dinamici di elementi. Questi selettori sono particolarmente utili per gli elementi interattivi, come i collegamenti ipertestuali. Si può desiderare l'aspetto dei collegamenti ipertestuali quando il puntatore del mouse si sposta su di essi, per attirare l'attenzione del visitatore.

Tornando alla nostra pagina d'esempio, potremmo rimuovere le sottolineature dai link nel menu in alto e mostrare una linea solo quando il puntatore del mouse si sposta sul link corrispondente. Per fare questo, scriviamo prima una regola per rimuovere la sottolineatura dai link nel menu in alto:

```
.menu.top a {
    text-decoration: none
}
```

Poi usiamo la pseudo-classe `:hover` su quegli stessi elementi per creare una regola CSS che verrà applicata solo quando il puntatore del mouse è sopra l'elemento corrispondente:

```
.menu.top a:hover {
    text-decoration: underline
}
```

Il selettori di pseudo-classe `:hover` accetta tutte le proprietà CSS delle regole CSS convenzionali. Altre pseudo-classi includono `:visited`, che corrisponde ai collegamenti ipertestuali che sono già

stati visitati, e `:focus`, che corrisponde agli elementi di un modulo che hanno ricevuto il focus.

Esercizi Guidati

- Supponiamo che una pagina HTML abbia un foglio di stile associato a essa, contenente le due seguenti regole:

```
p {  
    color: green;  
}  
  
p {  
    color: red;  
}
```

Che colore applicherà il browser al testo all'interno degli elementi p?

- Qual è la differenza tra l'uso del selettore ID `div#main` e `#main`?

- Quale selettore corrisponde a tutti gli elementi `p` utilizzati all'interno di un `div` con attributo ID `#main`?

- Qual è la differenza tra l'uso del selettore di classe `p.highlight` e `.highlight`?

Esercizi Esplorativi

1. Scrivi una singola regola CSS che cambi la proprietà `font-style` in `oblique`. La regola deve corrispondere solo agli elementi `a` che sono dentro `<div id="sidebar"></div>` o `<ul class="link">`.

2. Supponiamo che tu voglia cambiare lo stile degli elementi il cui attributo `class` è impostato su `article reference`, come in `<p class="article reference">`. Tuttavia, il selettore `.article .reference` non sembra alterare il loro aspetto. Perché il selettore non funziona come previsto?

3. Scrivi una regola CSS per cambiare la proprietà `color` in `red` di tutti i link visitati nella pagina.

Sommario

Questa lezione tratta di come usare i selettori CSS e come il browser decide quali stili applicare a ogni elemento. Essendo separati dal markup HTML, i CSS forniscono molti selettori per abbinare singoli elementi o gruppi di elementi nella pagina. La lezione passa attraverso i seguenti concetti e procedure:

- Stili a livello di pagina ed ereditarietà degli stili.
- Styling degli elementi per tipo.
- Usare l'ID dell'elemento e la classe come selettore.
- Selettori composti.
- Usare le pseudo-classi per stilizzare dinamicamente gli elementi.

Risposte agli Esercizi Guidati

1. Supponiamo che una pagina HTML abbia un foglio di stile associato a essa, contenente le due seguenti regole:

```
p {  
    color: green;  
}  
  
p {  
    color: red;  
}
```

Che colore applicherà il browser al testo all'interno degli elementi p?

Il colore red. Quando due o più selettori equivalenti hanno proprietà in conflitto, il browser sceglie l'ultimo.

2. Qual è la differenza tra l'uso del selettore ID div#main e #main?

Il selettore div#main corrisponde ad un elemento div che ha l'ID main, mentre il selettore #main corrisponde all'elemento che ha l'ID main, indipendentemente dal suo tipo.

Quale selettore corrisponde a tutti gli elementi p utilizzati all'interno di un div con attributo ID #main?

+ Il selettore #main p o div#main p.

1. Qual è la differenza tra l'uso del selettore di classe p.highlight e .highlight?

Il selettore p.highlight corrisponde solo agli elementi di tipo p che hanno la classe highlight, mentre il selettore .highlight corrisponde a tutti gli elementi che hanno la classe highlight, indipendentemente dal loro tipo.

Risposte agli Esercizi Esplorativi

1. Scrivi una singola regola CSS che cambi la proprietà `font-style` in `oblique`. La regola deve corrispondere solo agli elementi `a` che sono dentro `<div id="sidebar"></div>` o `<ul class="links">`.

```
#sidebar a, ul.links a {  
    font-style: oblique  
}
```

2. Supponiamo che tu voglia cambiare lo stile degli elementi il cui attributo `class` è impostato su `article reference`, come in `<p class="article reference">`. Tuttavia, il selettore `.article .reference` non sembra alterare il loro aspetto. Perché il selettore non funziona come previsto?

Il selettore `.article .reference` corrisponderà agli elementi aventi la classe `reference` che sono discendenti di elementi aventi la classe `article`. Per far corrispondere elementi che hanno entrambe le classi `article` e `reference`, il selettore dovrebbe essere `.article.reference` (senza lo spazio tra di loro).

3. Scrivi una regola CSS per cambiare la proprietà `color` in `red` di tutti i link visitati nella pagina.

```
a:visited {  
    color: red;  
}
```



033.3 Stili nei CSS

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 033.3

Peso

2

Arearie di Conoscenza Chiave

- Comprendere le proprietà fondamentali dei CSS
- Comprendere le unità comunemente usate nei CSS

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- px, %, em, rem, vw, vh
- color, background, background-*, font, font-*, text-*, list-style, line-height



033.3 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	033 Stile dei Contenuti con i CSS
Obiettivo:	033.3 Stili nei CSS
Lezione:	1 di 1

Introduzione

I CSS forniscono centinaia di proprietà che possono essere utilizzate per modificare l'aspetto degli elementi HTML. Solo i designer esperti riescono a ricordarne la maggior parte. Tuttavia, è pratico conoscere le proprietà di base che sono applicabili a qualsiasi elemento, così come alcune proprietà specifiche dell'elemento stesso. Questo capitolo tratta di alcune proprietà popolari che probabilmente userete.

Proprietà e Valori Comuni dei CSS

Molte proprietà CSS hanno lo stesso tipo di valore. I colori, per esempio, hanno lo stesso formato numerico sia che stiate cambiando il colore del carattere o il colore dello sfondo. Allo stesso modo, le unità disponibili per cambiare la dimensione del carattere sono anche usate per cambiare lo spessore di un bordo. Tuttavia, il formato del valore non è sempre unico. I colori, per esempio, possono essere inseriti in vari formati diversi, come vedremo in seguito.

Colori

Cambiare il colore di un elemento è probabilmente una delle prime cose che i designer imparano a

fare con i CSS. Puoi cambiare il colore del testo, il colore dello sfondo, il colore del bordo degli elementi e altro ancora.

Il valore di un colore nei CSS può essere scritto come un *color keyword* (cioè un nome di colore) o come un valore numerico che elenca ogni componente del colore stesso. Tutti i nomi comuni dei colori, come `black`, `white`, `red`, `purple`, `green`, `yellow`, e `blue` sono accettati come parole chiave di colore. L'intera lista accettata nei CSS è disponibile su [pagina web del W3C](#). Ma per avere un controllo più fine sul colore, puoi usare il valore numerico.

Parole Chiave di Colore

Useremo prima la parola chiave `color`, perché è più semplice. La proprietà `color` cambia il colore del testo nell'elemento corrispondente. Quindi per mettere tutto il testo della pagina in viola, potresti scrivere la seguente regola CSS:

```
* {  
    color: purple;  
}
```

Valori Numerici dei Colori

Anche se intuitivo, le parole chiave di colore non offrono la gamma completa di colori possibili nei display moderni. I web designer di solito sviluppano una tavolozza di colori che impiega colori personalizzati, assegnando valori specifici ai componenti rosso, verde e blu.

Ogni componente del colore è un numero binario a otto bit, che possiamo esprimere con un valore che va da 0 a 255. Tutti e tre i colori base devono essere specificati nella miscela di colori, e il loro ordine è sempre rosso, verde, blu. Quindi, per cambiare il colore di tutto il testo nella pagina in rosso usando la notazione RGB, usa `rgb(255, 0, 0)`:

```
* {  
    color: rgb(255, 0, 0);  
}
```

Un componente impostato su `0` indica che il colore di base corrispondente non viene utilizzato nella miscela di colori. Le percentuali possono essere utilizzate anche al posto dei numeri:

```
* {
  color: rgb(100%, 0%, 0%);
}
```

La notazione RGB si vede raramente se si utilizza un'applicazione di disegno per creare layout o semplicemente per sceglierne i colori. Piuttosto, è più comune vedere i colori nei CSS espressi come valori *esadecimale*. Anche i componenti del colore in esadecimale vanno da 0 a 255, ma in modo più conciso, iniziando con un simbolo hash # e utilizzando una lunghezza fissa di due cifre per tutti i componenti. Il valore minimo 0 è 00 e il valore massimo 255 è FF, quindi il colore rosso è #FF0000.

TIP

Se le cifre in ogni componente di un colore esadecimale si ripetono, la seconda cifra può essere omessa. Il colore rosso, per esempio, può essere scritto con una sola cifra per ogni componente: #F00.

La seguente lista mostra la notazione RGB ed esadecimale per alcuni colori di base:

Nome Colore	Notazione RGB	Valore Esadecimale
black	rgb(0, 0, 0)	#000000
white	rgb(255, 255, 255)	#FFFFFF
red	rgb(255, 0, 0)	#FF0000
purple	rgb(128, 0, 128)	#800080
green	rgb(0, 128, 0)	#008000
yellow	rgb(255, 255, 0)	#FFFF00
blue	rgb(0, 0, 255)	#0000FF

Le parole chiave di colore e le lettere nei valori esadecimali di colore sono *case-insensitive*.

Opacità di Colore

Oltre ai colori *saturi*, è possibile riempire gli elementi della pagina con colori *semitrasparenti*. L'opacità di un colore può essere impostata usando un quarto componente nel valore del colore. A differenza degli altri componenti del colore, dove i valori sono numeri interi che vanno da 0 a 255, l'opacità è una frazione che va da 0 a 1.

Il valore più basso, 0, dà come risultato un colore completamente trasparente, rendendolo indistinguibile da qualsiasi altro colore completamente trasparente. Il valore più alto, 1, dà come risultato un colore completamente saturo, che è lo stesso del colore originale senza alcuna trasparenza.

Quando usi la notazione RGB, specifica i colori con una componente di opacità attraverso il prefisso `rgba` invece di `rgb`. Così, per rendere il colore rosso semitrasparente, usate `rgba(255, 0, 0, 0.5)`. Il carattere `a` sta per *alpha channel*, un termine comunemente usato per specificare la componente di saturazione nel gergo della grafica digitale.

La saturazione può anche essere impostata nella notazione esadecimale. Qui, come per gli altri componenti del colore, la saturazione va da `00` a `FF`. Quindi, per rendere il colore rosso semitrasparente usando la notazione esadecimale, usa `#FF000080`.

Sfondo

Il colore di sfondo dei singoli elementi o dell'intera pagina è impostato con la proprietà `background-color`. Accetta gli stessi valori della proprietà `color`, sia come parole chiave che usando la notazione RGB/esadecimale.

Lo sfondo degli elementi HTML non è però limitato ai colori. Con la proprietà `background-image` è possibile usare un'immagine come sfondo. I formati di immagine accettati sono tutti quelli convenzionali accettati dal browser, come JPEG e PNG.

Il percorso dell'immagine dovrebbe essere specificato usando un designatore `url()`. Se l'immagine che vuoi usare è nella stessa cartella del file HTML, è sufficiente usare solo il nome del file:

```
body {  
    background-image: url("background.jpg");  
}
```

In questo esempio, il file immagine `background.jpg` sarà usato come immagine di sfondo per tutto il corpo della pagina. Per impostazione predefinita, l'immagine di sfondo viene ripetuta se la sua dimensione non copre l'intera pagina, partendo dall'angolo in alto a sinistra dell'area corrispondente al selettori della regola. Questo comportamento può essere modificato con la proprietà `background-repeat`. Se vuoi che l'immagine di sfondo sia posizionata nell'area dell'elemento senza essere ripetuta, usa il valore `no-repeat`:

```
body {  
    background-image: url("background.jpg");  
    background-repeat: no-repeat;  
}
```

Puoi anche far ripetere l'immagine solo in direzione orizzontale (`background-repeat: repeat-x`) o solo in direzione verticale (`background-repeat: repeat-y`).

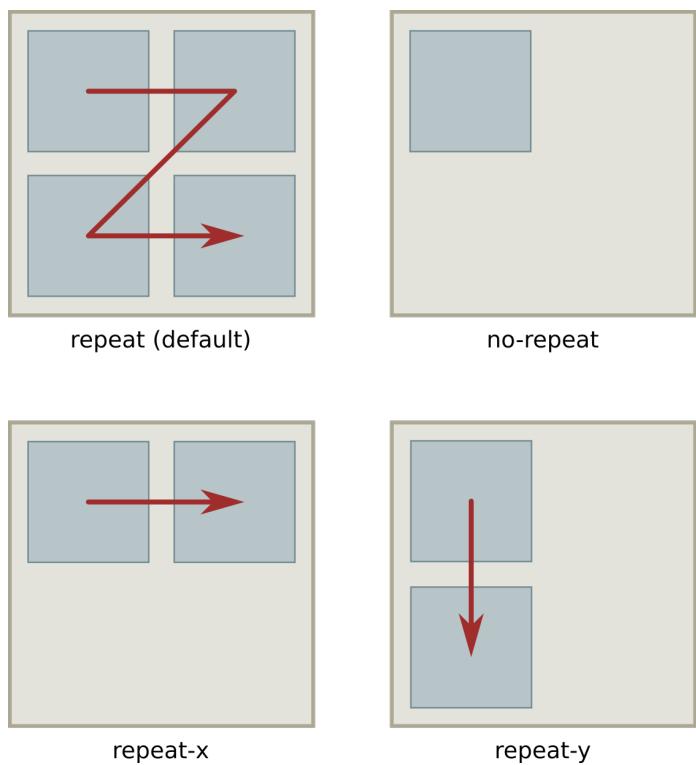


Figure 34. Posizionamento dello sfondo utilizzando la proprietà `background-repeat`.

TIP

Due o più proprietà CSS possono essere combinate in una singola proprietà, chiamata proprietà `background shorthand`. Le proprietà `background-image` e `background-repeat`, per esempio, possono essere combinate in una singola proprietà `background` con `background: no-repeat url("background.jpg")`.

Un'immagine di sfondo può anche essere posta in una posizione specifica nell'area dell'elemento usando la proprietà `background-position`. Le cinque posizioni di base sono `top`, `bottom`, `left`, `right` e `center`, ma la posizione in alto a sinistra dell'immagine può anche essere regolata con le percentuali:

```
body {
  background-image: url("background.jpg");
  background-repeat: no-repeat;
  background-position: 30% 10%;
}
```

La percentuale per ogni posizione è relativa alla dimensione corrispondente dell'elemento. Nell'esempio, il lato sinistro dell'immagine di sfondo sarà al 30% della larghezza del corpo (di solito il corpo è l'intero documento visibile) e il lato superiore dell'immagine sarà al 10% dell'altezza del corpo.

Bordi

Cambiare il bordo di un elemento è anche una comune personalizzazione del layout fatta con i CSS. Il bordo si riferisce alla linea che forma un rettangolo intorno all'elemento e ha tre proprietà di base: `color`, `style` e `width`.

Il colore del bordo, definito con `border-color`, segue lo stesso formato che abbiamo visto per le altre proprietà di colore.

I bordi possono essere tracciati in uno stile diverso da una linea continua. Potresti, per esempio, usare dei trattini per il bordo con la proprietà `border-style: dashed`. Gli altri possibili valori di stile sono:

dotted

Una sequenza di punti arrotondati

double

Due lineerette

groove

Una linea con un aspetto scolpito

ridge

Una linea con un aspetto espanso

inset

Un elemento che appare incorporato

outset

Un elemento che appare in rilievo

La proprietà `border-width` imposta lo spessore del bordo. Il suo valore può essere una parola chiave (`thin`, `medium` o `thick`) o un valore numerico specifico. Se preferisci usare un valore numerico, dovrà anche specificare la sua unità corrispondente. Questo aspetto sarà affrontato in seguito.

Valori di Unità

Le dimensioni e le distanze nei CSS possono essere definite in vari modi. Le unità assolute sono basate su un numero fisso di pixel dello schermo, quindi non sono molto diverse dalle dimensioni fisse usate nei media stampati. Ci sono anche unità relative, che sono calcolate dinamicamente rispetto al media dove la pagina viene visualizzata, come lo spazio disponibile o un'altra dimensione

scritta in unità assolute.

Unità Assolute

Le unità assolute sono equivalenti alle loro controparti fisiche, come i centimetri o i pollici. Sugli schermi dei computer convenzionali, un pollice sarà largo 96 pixel. Le seguenti unità assolute sono comunemente usate:

in (inch)

1 in è equivalente a 2,54 cm o 96 px.

cm (centimetri)

1 cm è equivalente a 96 px / 2.54.

mm (millimeter)

1 mm è equivalente a 1 cm / 10.

px (pixel)

1 px è equivalente a 1 / 96esimo di un pollice (inch).

pt (point)

1pt è equivalente a 1 / 72esimo di un pollice (inch).

Tieni presente che il rapporto tra pixel e pollici può variare. Negli schermi ad alta risoluzione, dove i pixel sono più densi, un pollice corrisponderà a più di 96 pixel.

Unità Relative

Le unità relative variano a seconda di altre misure o delle dimensioni della finestra. Il *viewport* è l'area del documento attualmente visibile nella sua finestra. In modalità schermo intero, la finestra corrisponde allo schermo del dispositivo. Le seguenti unità relative sono comunemente usate:

%

Percentuale – è relativo all'elemento genitore.

em

La dimensione del carattere usato nell'elemento.

rem

La dimensione del carattere usato nell'elemento radice.

vw

1% della larghezza della finestra.

vh

1% dell'altezza della finestra.

Il vantaggio di usare unità relative è che puoi creare layout che sono regolabili cambiando solo alcune dimensioni chiave. Per esempio, puoi usare l'unità `pt` per impostare la dimensione del carattere nell'elemento `body` e l'unità `rem` per i caratteri negli altri elementi. Una volta cambiata la dimensione del carattere per il corpo, tutte le altre dimensioni dei caratteri si regoleranno di conseguenza. Inoltre, l'uso di `vw` e `vh` per impostare le dimensioni delle sezioni della pagina le rende adattabili a schermi con dimensioni diverse.

Caratteri e Proprietà del Testo

La tipografia, o lo studio dei tipi di carattere, è un argomento di design molto ampio, e la tipografia nei CSS non è da meno. Tuttavia, ci sono alcune proprietà di base dei font che soddisferanno i bisogni della maggior parte degli utenti che imparano i CSS.

La proprietà `font-family` imposta il nome del carattere da usare. Non c'è garanzia che il font scelto sarà disponibile nel sistema in cui la pagina verrà visualizzata, quindi questa proprietà potrebbe non avere alcun effetto nel documento. Sebbene sia possibile fare in modo che il browser scarichi e usi il file di font specificato, la maggior parte dei web designer non si fa problema a utilizzare una famiglia di font generica nei loro documenti.

Le tre famiglie di caratteri generiche più comuni sono `serif`, `sans-serif` e `monospace`. `Serif` è la famiglia di caratteri predefinita nella maggior parte dei browser. Se preferisci usare `sans-serif` per l'intera pagina, aggiungi la seguente regola al tuo foglio di stile:

```
* {  
    font-family: sans-serif;  
}
```

Opzionalmente, è possibile impostare prima un nome specifico di famiglia di font, seguito da una famiglia generica:

```
* {  
    font-family: "DejaVu Sans", sans-serif;  
}
```

Se il dispositivo che visualizza la pagina ha quella specifica famiglia di caratteri, il browser la userà. Altrimenti, userà il suo font predefinito che corrisponde al nome generico della famiglia. I browser cercano i font nell'ordine in cui sono specificati nella proprietà.

Chiunque abbia usato un'applicazione di elaborazione testi avrà anche familiarità con altre tre regolazioni del carattere: dimensione, spessore e stile. Queste tre regolazioni hanno delle controparti nelle proprietà CSS: `font-size`, `font-weight` e `font-style`.

La proprietà `font-size` accetta parole chiave come `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, `xxx-large`. Queste parole chiave sono relative alla dimensione predefinita del carattere usata dal browser. Le parole chiave `larger` e `smaller` cambiano la dimensione del carattere rispetto alla dimensione del carattere dell'elemento principale. Puoi anche esprimere la dimensione del carattere con valori numerici, includendo l'unità dopo il valore, o con percentuali.

Se non vuoi cambiare la dimensione del carattere, ma la distanza tra le linee, usa la proprietà `line-height`. Un `line-height` di 1 renderà l'altezza della linea la stessa dimensione del carattere dell'elemento, il che può rendere le linee di testo troppo vicine tra loro. Pertanto, un valore maggiore di 1 è più appropriato per i testi. Come la proprietà `font-size`, altre unità possono essere usate insieme al valore numerico.

Il `font-weight` imposta lo spessore del carattere con le familiari parole chiave `normal` o `bold`. Le parole chiave `lighter` e `bolder` cambiano lo spessore del carattere dell'elemento rispetto a quello del carattere del suo elemento principale.

La proprietà `font-style` può essere impostata su `italic` per selezionare la versione in corsivo della famiglia di caratteri corrente. Il valore `oblique` seleziona la versione obliqua del font. Queste due opzioni sono quasi identiche, ma la versione italica di un font è solitamente un po' più stretta della versione obliqua. Se non esistono né versioni italiche né oblique del font, il font sarà artificialmente inclinato dal browser.

Ci sono altre proprietà che cambiano il modo in cui il testo viene reso nel documento. Puoi, per esempio, aggiungere una sottolineatura ad alcune parti del testo che vuoi enfatizzare. Per prima cosa, usa un tag `` per delimitare il testo:

`<p>CSS is the proper mechanism to style HTML documents.</p>`

Poi puoi usare il selettori `.under` per cambiare la proprietà `text-decoration`:

```
.under {  
    text-decoration: underline;  
}
```

Per default, tutti gli elementi `a` (link) sono sottolineati. Se vuoi rimuoverlo, usa il valore `none` per la `text-decoration` di tutti gli elementi `a`:

```
a {  
    text-decoration: none;  
}
```

Quando si revisiona il contenuto, alcuni autori preferiscono cancellare parti errate o obsolete del testo, in modo che il lettore sappia che il testo è stato aggiornato e che cosa è stato rimosso. Per farlo, usa il valore `line-through` della proprietà `text-decoration`:

```
.disregard {  
    text-decoration: line-through;  
}
```

Di nuovo, un tag `` può essere usato per applicare lo stile:

```
<p>Netscape Navigator <span class="disregard">is</span> was one of the most popular  
Web browsers.</p>
```

Altre decorazioni possono essere specifiche per un elemento. I pallini negli elenchi puntati possono essere personalizzati usando la proprietà `list-style-type`. Per cambiarli in quadrati, per esempio, usa `list-style-type: square`. Per rimuoverli semplicemente, imposta il valore di `list-style-type` a `none`.

Esercizi Guidati

1. Come si può aggiungere la semitrasparenza al colore blu (notazione RGB `rgb(0,0,255)`) per usarlo nella proprietà CSS `color`?

2. Quale colore corrisponde al valore esadecimale `#000`?

3. Dato che `Times New Roman` è un font serif e che non sarà disponibile in tutti i dispositivi, come potresti scrivere una regola CSS per richiedere `Times New Roman` mentre imposti la famiglia di font generica serif come *fallback*?

4. Come potresti usare una parola chiave di dimensione relativa per impostare la dimensione del carattere dell'elemento `<p class="disclaimer">` più piccolo in relazione al suo elemento principale?

Esercizi Esplorativi

1. La proprietà `background` è un'abbreviazione per impostare più di una proprietà `background-*` contemporaneamente. Riscrivi la seguente regola CSS come una singola proprietà `background`.

```
body {  
    background-image: url("background.jpg");  
    background-repeat: repeat-x;  
}
```

2. Scrivi una regola CSS per l'elemento `<div id="header"></div>` che cambia *solo* la larghezza del suo bordo inferiore a `4px`.

3. Scrivi una proprietà `font-size` che raddoppia la dimensione del carattere usato nell'elemento principale della pagina.

4. La *Doppia spaziatura* è una caratteristica comune di formattazione del testo nei word processor. Come si potrebbe impostare un formato simile usando i CSS?

Sommario

Questa lezione tratta l'applicazione di stili semplici agli elementi di un documento HTML. I CSS forniscono centinaia di proprietà, e la maggior parte dei web designer avrà bisogno di ricorrere a manuali di riferimento per utilizzarle tutte. Ciononostante, un insieme relativamente piccolo di proprietà e valori sono usati la maggior parte delle volte, ed è importante conoscerli a memoria. La lezione passa attraverso i seguenti concetti e procedure:

- Proprietà CSS fondamentali che hanno a che fare con colori, sfondi e caratteri.
- Le unità assolute e relative che i CSS possono usare per impostare dimensioni e distanze, come px, em, rem, vw, vh e così via.

Risposte agli Esercizi Guidati

1. Come si può aggiungere la semitrasparenza al colore blu (notazione RGB `rgb(0,0,255)`) per usarlo nella proprietà CSS `color`?

Usa il prefisso `rgba` e aggiungi `0.5` come quarto valore: `rgba(0,0,0,0.5)`.

2. Quale colore corrisponde al valore esadecimale `#000`?

Il colore `black`. Il valore esadecimale `#000` è un'abbreviazione di `#000000`.

3. Dato che `Times New Roman` è un font `serif` e che non sarà disponibile in tutti i dispositivi, come potresti scrivere una regola CSS per richiedere `Times New Roman` mentre imposti la famiglia di font generica `serif` come `fallback`?

```
* {  
    font-family: "Times New Roman", serif;  
}
```

4. Come potresti usare una parola chiave di dimensione relativa per impostare la dimensione del carattere dell'elemento `<p class="disclaimer">` più piccolo in relazione al suo elemento principale?

Utilizzando la parola chiave `smaller`:

```
p.disclaimer {  
    font-size: smaller;  
}
```

Risposte agli Esercizi Esplorativi

1. La proprietà `background` è un'abbreviazione per impostare più di una proprietà `background-*` contemporaneamente. Riscrivi la seguente regola CSS come una singola proprietà `background`.

```
body {
  background-image: url("background.jpg");
  background-repeat: repeat-x;
}
```

```
body {
  background: repeat-x url("background.jpg");
}
```

2. Scrivi una regola CSS per l'elemento `<div id="header"></div>` che cambia *solo* la larghezza del suo bordo inferiore a 4px.

```
#header {
  border-bottom-width: 4px;
}
```

3. Scrivi una proprietà `font-size` che raddoppia la dimensione del carattere usato nell'elemento principale della pagina.

L'unità `rem` corrisponde alla dimensione del carattere usato nell'elemento radice, quindi la proprietà dovrebbe essere `font-size: 2rem`.

4. La *Doppia spaziatura* è una caratteristica comune di formattazione del testo nei word processor. Come si potrebbe impostare un formato simile usando i CSS?

Imposta la proprietà `line-height` al valore `2em`, perché l'unità `em` corrisponde alla dimensione del carattere dell'elemento corrente.



033.4 Modellazione e Disposizione dei Contenitori nei CSS

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 033.4

Peso

2

Arese di Conoscenza Chiave

- Definire la dimensione, la posizione e l'allineamento degli elementi in un layout CSS
- Specificare come il testo scorre intorno ad altri elementi
- Comprendere il flusso del documento
- Conoscenza della griglia CSS
- Conoscenza del responsive web design
- Conoscenza delle media query CSS

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- width, height, padding, padding-*, margin, margin-*, border, border-*
- top, left, right, bottom
- display: block | inline | flex | inline-flex | none
- position: static | relative | absolute | fixed | sticky
- float: left | right | none
- clear: left | right | both | none



033.4 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	033 Stile dei Contenuti con i CSS
Obiettivo:	033.4 Modellazione e Disposizione dei Contenitori nei CSS
Lezione:	1 di 1

Introduzione

Ogni elemento visibile in un documento HTML è reso come un contenitore rettangolare (*box*). Così, il termine *box model* descrive l'approccio che i CSS hanno nel modificare le proprietà visuali degli elementi. Come contenitori di diverse dimensioni, gli elementi HTML possono essere annidati all'interno di elementi *container*—di solito l'elemento `div`—così da essere organizzati in sezioni e sotto-sezioni.

Possiamo usare i CSS per modificare la posizione dei “contenitori”, da piccoli aggiustamenti a drastici cambiamenti nella disposizione degli elementi nella pagina. Oltre al normale flusso, la posizione di ogni casella può essere basata sugli elementi che la circondano, o sulla sua relazione con il contenitore principale o sulla sua relazione con la *viewport*, che è l'area della pagina visibile all'utente. *Nessun* singolo meccanismo soddisfa tutti i possibili requisiti di layout, quindi può essere necessaria una combinazione di essi.

Flusso Normale

Il modo predefinito in cui il browser mostra l'albero dei documenti è chiamato *flusso normale*. I

rettangoli corrispondenti agli elementi sono posizionati più o meno nello stesso ordine in cui appaiono nell'albero del documento, rispetto ai loro elementi principali. Tuttavia, a seconda del tipo di elemento, il riquadro corrispondente può seguire regole di posizionamento distinte.

Un buon modo per capire la logica del flusso normale è rendere *visibili* i contenitori. Possiamo iniziare con una pagina molto semplice, con solo tre elementi `div` separati, ognuno dei quali ha un paragrafo con testo casuale:

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>CSS Box Model and Layout</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

<div id="first">
  <h2>First div</h2>
  <p><span>Sed</span> <span>eget</span> <span>velit</span>
    <span>id</span> <span>ante</span> <span>tempus</span>
    <span>porta</span> <span>pulvinar</span> <span>et</span>
    <span>ex.</span></p>
</div><!-- #first -->

<div id="second">
  <h2>Second div</h2>
  <p><span>Fusce</span> <span>vitae</span> <span>vehicula</span>
    <span>neque.</span> <span>Etiam</span> <span>maximus</span>
    <span>vulputate</span> <span>neque</span> <span>eu</span>
    <span>lobortis.</span> <span>Phasellus</span> <span>condimentum,</span>
    <span>felis</span> <span>eget</span> <span>eleifend</span>
    <span>aliquam,</span> <span>dui</span> <span>dolor</span>
    <span>bibendum</span> <span>leo.</span></p>
</div><!-- #second -->

<div id="third">
  <h2>Third div</h2>
  <p><span>Pellentesque</span> <span>ornare</span> <span>ultricies</span>
    <span>elementum.</span> <span>Morbi</span> <span>vulputate</span>
    <span>premium</span> <span>arcu,</span> <span>sed</span>
    <span>faucibus.</span></p>
</div><!-- #third -->

</body>
</html>

```

Ogni parola è in un elemento `span`, così possiamo dare uno stile alle parole e vedere come vengono trattate anche come caselle. Per rendere visibili i contenitori, dobbiamo modificare il file del foglio di stile `style.css` cui fa riferimento il documento HTML. Le seguenti regole implementeranno ciò di cui abbiamo bisogno:

```
* {
    font-family: sans;
    font-size: 14pt;
}

div {
    border: 2px solid #00000044;
}

#first {
    background-color: #c4a000ff;
}

#second {
    background-color: #4e9a06ff;
}

#third {
    background-color: #5c3566da;
}

h2 {
    background-color: #fffff66;
}

p {
    background-color: #fffff66;
}

span {
    background-color: #fffffaa;
}
```

Il risultato viene mostrato in [Figure 35](#).

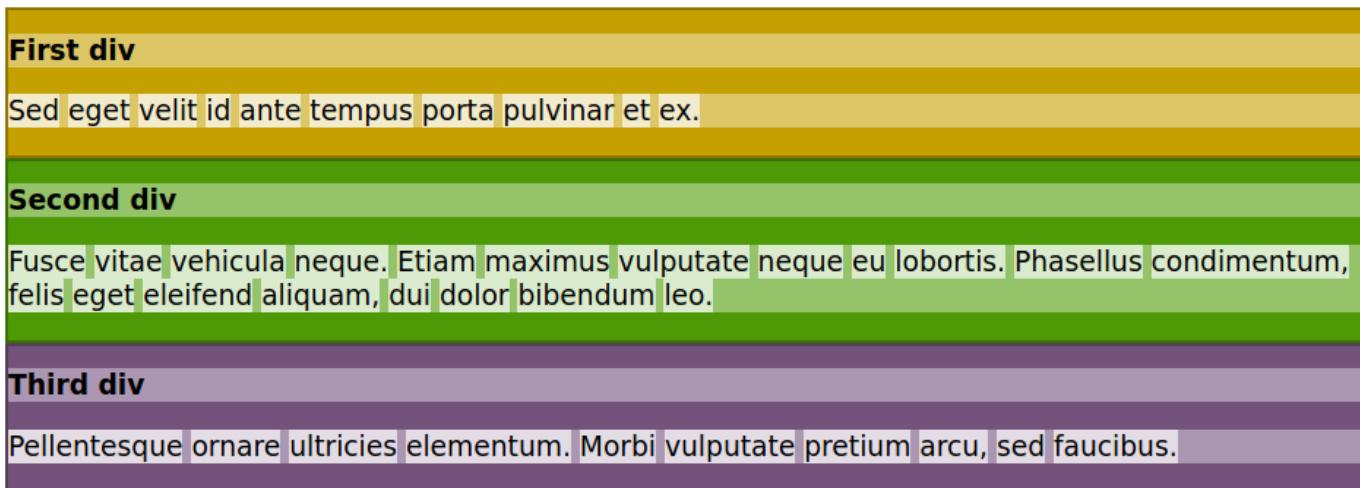


Figure 35. Il flusso degli elementi di base è dall'alto in basso e da sinistra a destra.

Figure 35 mostra che ogni tag HTML ha una casella corrispondente nel layout. Gli elementi `div`, `h2` e `p` si estendono alla larghezza del loro elemento principale. Per esempio, l'elemento genitore degli elementi `div` è l'elemento `body`, quindi si estendono alla larghezza del corpo, mentre il genitore di ogni elemento `h2` e `p` è il suo corrispondente `div`. I contenitori che si estendono fino alla larghezza del loro elemento principale sono chiamati elementi *block*. Alcuni dei più comuni tag HTML resi come blocchi sono `h1`, `h2`, `h3`, `p`, `ul`, `ol`, `table`, `li`, `div`, `section`, `form`, e `aside`. Gli elementi di blocco “gemelli” – elementi di blocco che condividono lo stesso elemento genitore – sono impilati all'interno del loro genitore dall'alto verso il basso.

NOTE

Alcuni elementi di blocco non sono pensati per essere usati come contenitori di altri elementi di blocco. È possibile, per esempio, inserire un elemento di blocco all'interno di un elemento `h1` o `p`, ma non è considerato una buona pratica. Piuttosto, il designer dovrebbe usare un tag appropriato come contenitore. I tag contenitore comuni sono `div`, `section` e `aside`.

Oltre al testo stesso, elementi come `h1`, `p` e `li` si aspettano solo elementi *inline* come figli. Come la maggior parte dei testi occidentali, gli elementi inline seguono il flusso di testo da sinistra a destra. Quando non c'è più spazio nella parte destra, il flusso degli elementi inline continua nella riga successiva, proprio come il testo. Alcuni tag HTML comuni trattati come contenitori inline sono `span`, `a`, `em`, `strong`, `img`, `input` e `label`.

Nella nostra pagina HTML di esempio, ogni parola all'interno dei paragrafi era inserita all'interno di un tag `span`, in modo da poter essere evidenziata con una regola CSS corrispondente. Come mostrato nell'immagine, ogni elemento `span` è posizionato orizzontalmente, da sinistra a destra, finché non c'è più spazio nell'elemento principale.

L'altezza dell'elemento dipende dal suo contenuto, così il browser regola l'altezza di un elemento contenitore per ospitare i suoi elementi a blocchi annidati o le linee di elementi in linea. Tuttavia, alcune proprietà CSS influenzano la forma di un contenitore, la sua posizione e il posizionamento dei suoi elementi interni.

Le proprietà `margin` e `padding` influenzano tutti i tipi di casella. Se non si impostano esplicitamente queste proprietà, il browser ne imposta alcune utilizzando valori standard. Come visto in [Figure 35](#), gli elementi `h2` e `p` sono stati resi con uno spazio tra di loro. Questi spazi sono i margini superiori e inferiori che il browser aggiunge di default a questi elementi. Possiamo rimuoverli modificando le regole CSS per i selettori `h2` e `p`:

```
h2 {
    background-color: #fffff66;
    margin: 0;
}

p {
    background-color: #fffff66;
    margin: 0;
}
```

Il risultato appare in [Figure 36](#).

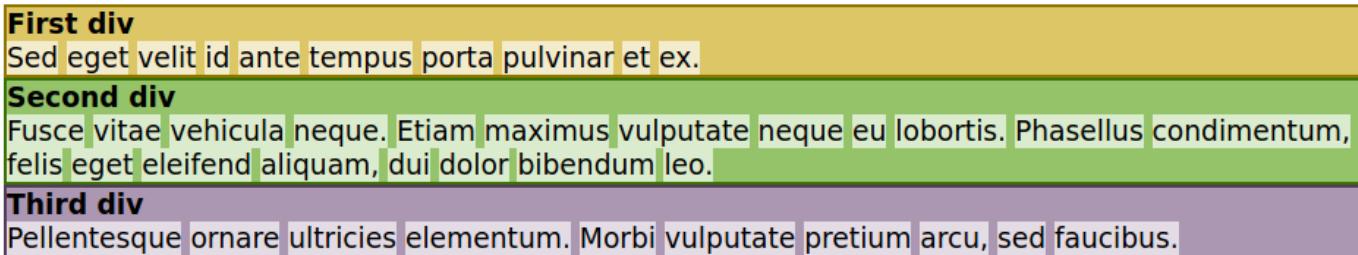


Figure 36. La proprietà margin può cambiare o rimuovere i margini dagli elementi.

L'elemento `body` ha anche, per impostazione predefinita, un piccolo margine che crea uno spazio circostante. Anche questo spazio può essere rimosso usando la proprietà `margin`.

Mentre la proprietà `margin` definisce lo spazio tra l'elemento e ciò che c'è intorno, la proprietà `padding` dell'elemento definisce lo spazio interno tra i limiti del contenitore e i suoi elementi figli. Considera gli elementi `h2` e `p` all'interno di ogni `div` nel codice di esempio. Potremmo usare la loro proprietà `margin` per creare uno spazio ai bordi del corrispondente `div`, ma è più semplice cambiare la proprietà `padding` del contenitore:

```
#second {
    background-color: #4e9a06ff;
    padding: 1em;
}
```

Solo la regola per il secondo `div` è stata modificata, quindi i risultati ([Figure 37](#)) mostrano la differenza tra il secondo `div` e gli altri contenitori `div`.

The screenshot shows three `div` elements with different padding values:

- First div**: Yellow background, no padding.
- Second div**: Green background, padding of 1em.
- Third div**: Purple background, no padding.

The text content of the second `div` is longer than the first and third, demonstrating how padding affects the available space.

Figure 37. Diversi contenitori `div` possono avere padding diversi.

La proprietà `margin` è un'abbreviazione per quattro proprietà che controllano i quattro lati del contenitore: `margin-top`, `margin-right`, `margin-bottom` e `margin-left`. Quando a `margin` è assegnato un singolo valore, come negli esempi usati finora, tutti e quattro i margini del box lo usano. Quando sono scritti due valori, il primo definisce i margini superiore e inferiore, mentre il secondo definisce i margini destro e sinistro. Usando `margin: 1em 2em`, per esempio, si definisce uno spazio di 1 em per i margini superiore e inferiore e uno spazio di 2 em per i margini destro e sinistro. Scrivere quattro valori imposta i margini per i quattro lati in senso orario, iniziando dall'alto. I diversi valori nella proprietà non sono tenuti a usare le stesse unità.

Anche la proprietà `padding` è un'abbreviazione che segue gli stessi principi della proprietà `margin`.

Nel loro comportamento predefinito, gli elementi del blocco si allungano per adattarsi alla larghezza disponibile. Ma questo non è obbligatorio. La proprietà `width` può impostare una dimensione orizzontale fissa per il contenitore:

```
#first {
    background-color: #c4a000ff;
    width: 6em;
}
```

L'aggiunta di `width: 6em` alla regola CSS restringe il primo `div` orizzontalmente, lasciando uno spazio vuoto alla sua destra ([Figure 38](#)).

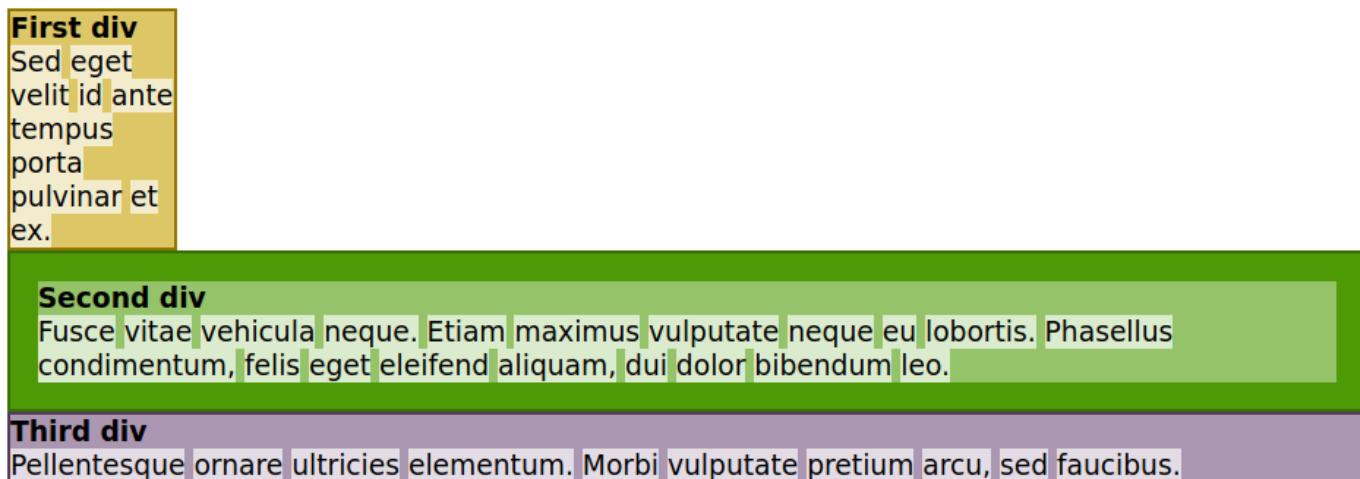


Figure 38. La proprietà width cambia la larghezza orizzontale del primo div.

Invece di lasciare il primo div allineato a sinistra, potremmo volerlo centrare. Centrare un box equivale a impostare margini della stessa dimensione su entrambi i lati, quindi possiamo usare la proprietà margin per centrarlo. La dimensione dello spazio disponibile può variare, quindi usiamo il valore auto per i margini destro e sinistro:

```
#first {  
    background-color: #c4a000ff;  
    width: 6em;  
    margin: 0 auto;  
}
```

I margini sinistro e destro sono calcolati automaticamente dal browser e il contenitore sarà centrato ([Figure 39](#)).

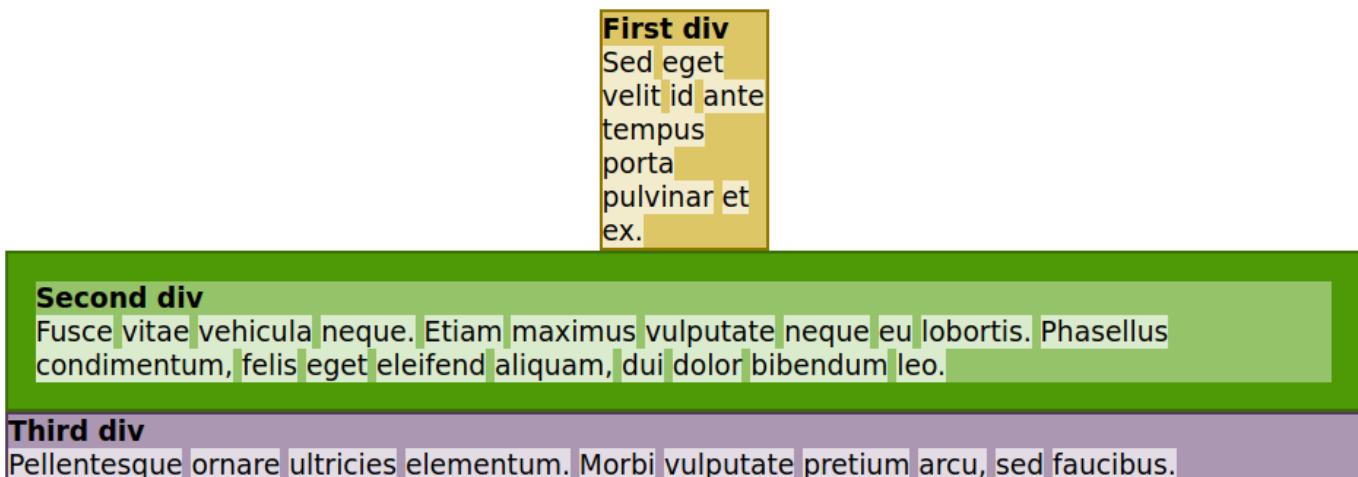


Figure 39. La proprietà margin è usata per centrare il primo div.

Come mostrato, rendere più stretto un elemento del blocco non rende lo spazio rimanente disponibile per l'elemento successivo. Il flusso naturale è ancora conservato, come se l'elemento più stretto occupasse ancora tutta la larghezza disponibile.

Personalizzazione del Flusso Normale

Il flusso normale è semplice e sequenziale. I CSS ti permettono anche di interrompere il flusso normale e posizionare gli elementi in modi molto specifici, anche scavalcando lo scorrimento della pagina. Vedremo diversi modi per controllare il posizionamento degli elementi in questa sezione.

Elementi Flottanti

È possibile fare in modo che gli elementi dei blocchi gemelli condividano lo stesso spazio orizzontale. Un modo per farlo è attraverso la proprietà `float`, che rimuove l'elemento dal flusso normale. Come suggerisce il nome, la proprietà `float` rende il riquadro flottante sopra gli elementi di blocco che vengono dopo, così essi saranno resi come se fossero sotto il riquadro flottante. Per rendere il primo div flottante a destra, aggiungi `float: right` alla regola CSS corrispondente:

```
#first {
  background-color: #c4a000ff;
  width: 6em;
  float: right;
}
```

I margini automatici sono ignorati in un contenitore flottante, quindi la proprietà `margin` può essere rimossa. Figure 40 mostra il risultato del *floating* del primo div a destra.

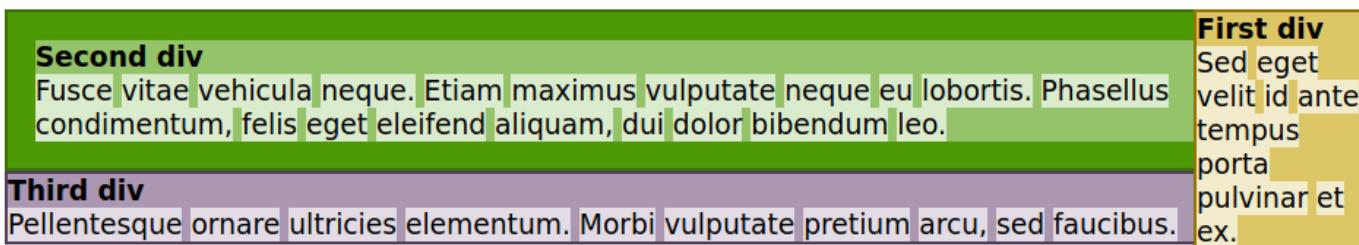


Figure 40. Il primo div è flottante e non fa parte del flusso normale.

Per impostazione predefinita tutti gli elementi del blocco che vengono dopo l'elemento flottante andranno sotto di esso. Pertanto, data un'altezza sufficiente, il riquadro flottante coprirà tutti i restanti elementi del blocco.

Anche se un elemento flottante va sopra altri elementi a blocchi, il contenuto in linea all'interno del contenitore dell'elemento flottante si avvolge intorno all'elemento stesso: come in riviste e giornali, che spesso avvolgono il testo intorno a un'immagine.

L'immagine precedente mostra come il primo div copra il secondo div e parte del terzo. Supponiamo di voler far fluttuare il primo div sopra il secondo div, ma non sopra il terzo. La soluzione è includere la proprietà `clear` nella regola CSS corrispondente al terzo div:

```
#third {
    background-color: #5c3566da;
    clear: right;
}
```

Impostando la proprietà `clear` a `right`, l'elemento corrispondente salta qualsiasi elemento precedente spostato a destra, riprendendo il flusso normale (Figure 41).

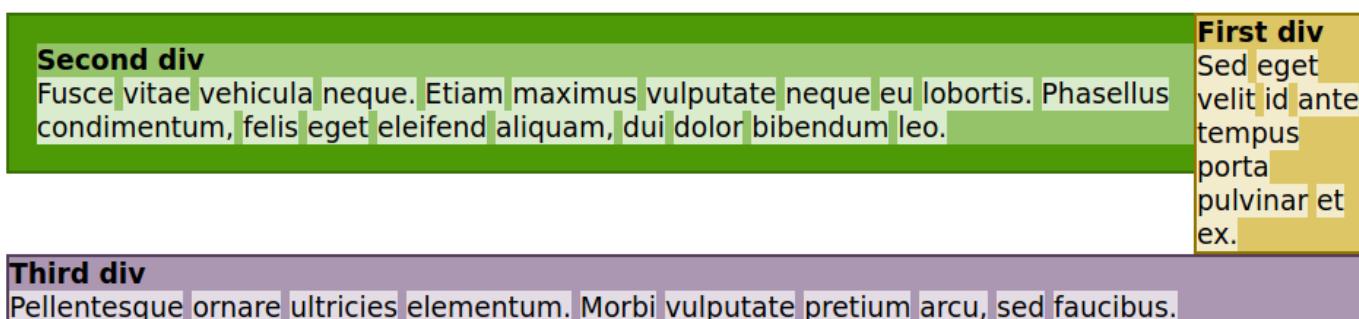


Figure 41. La proprietà clear ritorna al flusso normale.

Allo stesso modo, se un elemento precedente flotta a sinistra, puoi usare `clear: left` per riprendere il flusso normale. Quando devi saltare elementi flottanti sia a sinistra sia a destra, usa `clear: both`.

Posizionare Contenitori

Nel flusso normale ogni casella va dopo le caselle che la precedono nell'albero del documento. Gli elementi fratelli precedenti “spingono” gli elementi che vengono dopo di loro, spostandoli a destra e in basso all'interno del loro elemento principale. L'elemento genitore può avere i propri fratelli che avranno lo stesso comportamento. È come mettere le piastrelle una accanto all'altra in un muro, cominciando dall'alto.

Questo metodo di posizionamento dei contenitori è chiamato *static*, ed è il valore predefinito per la proprietà CSS `position`. A parte la definizione dei margini e del `padding`, *non c'è* modo di riposizionare un contenitore statico nella pagina.

Come l'analogia delle piastrelle sul muro, il posizionamento statico non è obbligatorio. Come per le piastrelle, puoi posizionare le caselle ovunque tu voglia, anche coprendo altre caselle. Per farlo, assegna alla proprietà `position` uno dei seguenti valori:

relative

L'elemento segue il normale flusso del documento, ma può usare le proprietà `top`, `right`, `bottom` e `left` per impostare degli *offset* rispetto alla sua posizione statica originale. Gli offset possono anche essere negativi. Gli altri elementi rimangono nelle loro posizioni originali, come se l'elemento relativo fosse ancora statico.

absolute

L'elemento ignora il normale flusso degli altri elementi e si posiziona nella pagina tramite le proprietà `top`, `right`, `bottom`, e `left`. I loro valori sono relativi al corpo del documento o a un contenitore principale non statico.

fixed

L'elemento ignora il normale flusso degli altri elementi e si posiziona tramite le proprietà `top`, `right`, `bottom`, e `left`. I valori sono relativi al *viewport* (cioè l'area dello schermo dove viene mostrato il documento). Gli elementi fissi non si muovono mentre il visitatore scorre il documento, ma assomigliano a un adesivo fissato sullo schermo.

sticky

L'elemento segue il normale flusso del documento. Tuttavia, invece di uscire dal *viewport* quando il documento scorre, si fermerà nella posizione impostata dalle proprietà `top`, `right`, `bottom` e `left`. Se il valore `top` è `10px`, per esempio, l'elemento smetterà di scorrere sotto la parte superiore del *viewport* quando raggiunge 10 pixel dal limite superiore del *viewport*. Quando ciò accade, il resto della pagina continua a scorrere, ma l'elemento adesivo si comporta come un elemento fisso in quella posizione. Tornerà alla sua posizione originale quando il documento scorrerà di nuovo nella sua posizione nel *viewport*. Gli elementi adesivi sono comunemente usati al giorno d'oggi

per creare menu superiori che siano sempre visibili.

Le posizioni che possono usare le proprietà `top`, `right`, `bottom` e `left` non sono obbligate a usarle tutte. Se imposta entrambe le proprietà `top` e `height` di un elemento assoluto, per esempio, il browser calcola implicitamente la sua proprietà `bottom` (`top + height = bottom`).

La Proprietà `display`

Se l'ordine dato dal flusso normale non è un problema nel tuo progetto, ma vuoi cambiare l'allineamento dei vari contenitori nella pagina, modifica la proprietà `display` dell'elemento. La proprietà `display` può anche far scomparire completamente l'elemento dal documento visualizzato, impostando `display: none`. Questo è utile quando si vuole mostrare l'elemento in seguito usando JavaScript.

La proprietà `display` può anche, per esempio, far comportare un elemento a blocchi come un elemento in linea (`display: inline`). Questa non è però considerata una buona pratica. Esistono metodi migliori per mettere gli elementi contenitore uno accanto all'altro, come il modello `flexbox`.

Il modello `flexbox` è stato inventato per superare le limitazioni dei `float` e per eliminare l'uso inappropriate delle tabelle per strutturare il layout della pagina. Quando si impone la proprietà `display` di un elemento contenitore su `flex` per trasformarlo in un contenitore `flexbox`, i suoi figli più prossimi si comporteranno più o meno come celle in una riga di tabella.

TIP

Se vuoi ancora più controllo sul posizionamento degli elementi nella pagina, dai un'occhiata alla funzione *CSS grid*. La griglia è un potente sistema basato su righe e colonne per creare layout elaborati.

Per testare la visualizzazione `flex`, aggiungi un nuovo elemento `div` alla pagina di esempio e rendilo il contenitore dei tre elementi `div` esistenti:

```

<div id="container">

  <div id="first">
    <h2>First div</h2>
    <p><span>Sed</span> <span>eget</span> <span>velit</span>
      <span>id</span> <span>ante</span> <span>tempus</span>
      <span>porta</span> <span>pulvinar</span> <span>et</span>
      <span>ex.</span></p>
  </div><!-- #first -->

  <div id="second">
    <h2>Second div</h2>
    <p><span>Fusce</span> <span>vitae</span> <span>vehicula</span>
      <span>neque.</span> <span>Etiam</span> <span>maximus</span>
      <span>vulputate</span> <span>neque</span> <span>eu</span>
      <span>lobortis.</span> <span>Phasellus</span> <span>condimentum,</span>
      <span>felis</span> <span>eget</span> <span>eleifend</span>
      <span>aliquam,</span> <span>dui</span> <span>dolor</span>
      <span>bibendum</span> <span>leo.</span></p>
  </div><!-- #second -->

  <div id="third">
    <h2>Third div</h2>
    <p><span>Pellentesque</span> <span>ornare</span> <span>ultricies</span>
      <span>elementum.</span> <span>Morbi</span> <span>vulputate</span>
      <span>pretium</span> <span>arcu,</span> <span>sed</span>
      <span>faucibus.</span></p>
  </div><!-- #third -->

</div><!-- #container -->

```

Aggiungi la seguente regola CSS al foglio di stile per trasformare il contenitore `div` in un contenitore flexbox:

```

#container {
  display: flex;
}

```

Il risultato sono i tre elementi interni `div` mostrati fianco a fianco ([Figure 42](#)).

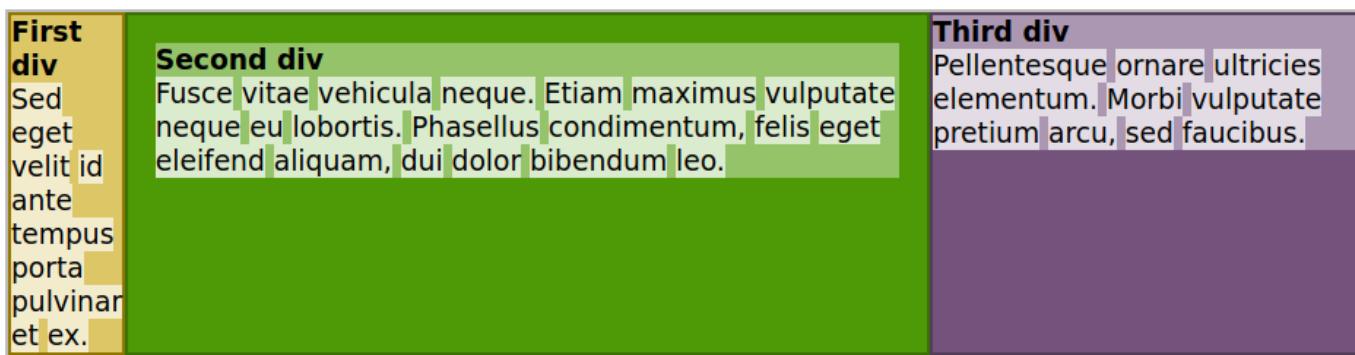


Figure 42. Il modello flexbox crea una griglia.

Usare il valore `inline-flex` invece di `flex` ha fondamentalmente lo stesso risultato, ma fa sì che i figli si comportino più come elementi *inline*.

Design Responsivo

Sappiamo che i CSS forniscono proprietà che regolano le dimensioni degli elementi e dei caratteri in relazione all'area dello schermo disponibile. Tuttavia, potresti voler andare oltre e usare un design diverso per dispositivi diversi: per esempio, sistemi *desktop* rispetto a dispositivi con schermi di dimensioni inferiori a una certa misura. Questo approccio è chiamato *responsive web design*, e i CSS forniscono metodi chiamati *media queries* per renderlo possibile.

Nell'esempio precedente, abbiamo modificato il layout della pagina per mettere gli elementi `div` uno accanto all'altro in colonne. Questo layout è adatto a schermi più grandi, ma sarà troppo ingombrante in schermi più piccoli. Per risolvere questo problema, possiamo aggiungere una media query al foglio di stile che corrisponda solo agli schermi con almeno `600px` di larghezza:

```
@media (min-width: 600px){
  #container {
    display: flex;
  }
}
```

Le regole CSS all'interno della direttiva `@media` saranno usate solo se il criterio tra parentesi è soddisfatto. In questo esempio, se la larghezza della finestra è inferiore a `600px`, la regola non sarà applicata al contenitore `div` e i suoi figli saranno resi come elementi convenzionali `div`. Il browser rivaluta le media query ogni volta che la dimensione della finestra cambia, così il layout può essere cambiato in tempo reale mentre si ridimensiona la finestra del browser o si ruota lo smartphone.

Esercizi Guidati

1. Se la proprietà `position` non viene modificata, quale metodo di posizionamento verrà utilizzato dal browser?

2. Come ci si può assicurare che il riquadro di un elemento sia visualizzato dopo qualsiasi elemento flottante?

3. Come puoi usare la proprietà stenografica `margin` per impostare i margini top/bottom a `4px` e i margini right/left a `6em`?

4. Come si può centrare orizzontalmente un elemento contenitore statico con larghezza fissa sulla pagina?

Esercizi Esplorativi

1. Scrivi una regola CSS che corrisponda all'elemento `<div class="picture">` in modo che il testo all'interno dei suoi susseguenti elementi di blocco si posizioni verso il suo lato destro.

2. Come influisce la proprietà `top` su un elemento statico rispetto al suo elemento principale?

3. In che modo cambiare la proprietà `display` di un elemento in `flex` influenza il suo posizionamento nel flusso normale?

4. Quale caratteristica dei CSS ti permette di usare un set separato di regole a seconda delle dimensioni dello schermo?

Sommario

Questa lezione si occupa della modellazione dei contenitori nei CSS e come possiamo personalizzarla. Oltre al normale flusso del documento, il designer può fare uso di diversi meccanismi di posizionamento per implementare un layout personalizzato. La lezione passa attraverso i seguenti concetti e procedure:

- Il normale flusso del documento.
- Regolazioni del margine e del bordo del contenitore di un elemento.
- Utilizzo delle proprietà float e clear.
- Meccanismi di posizionamento: static, relative, absolute, fixed e sticky.
- Valori alternativi per la proprietà display.
- Nozioni di base sul *responsive design*.

Risposte agli Esercizi Guidati

1. Se la proprietà `position` non viene modificata, quale metodo di posizionamento verrà utilizzato dal browser?

Il metodo `static`.

2. Come ci si può assicurare che il riquadro di un elemento sia visualizzato dopo qualsiasi elemento flottante?

La proprietà `clear` dell'elemento dovrebbe essere impostata su `both`.

3. Come puoi usare la proprietà stenografica `margin` per impostare i margini top/bottom a `4px` e i margini right/left a `6em`?

Può essere sia `margin: 4px 6em` sia `margin: 4px 6em 4px 6em`.

4. Come si può centrare orizzontalmente un elemento contenitore statico con larghezza fissa sulla pagina?

Usando il valore `auto` nelle sue proprietà `margin-left` e `margin-right`.

Risposte agli Esercizi Esplorativi

1. Scrivi una regola CSS che corrisponda all'elemento `<div class="picture">` in modo che il testo all'interno dei suoi susseguenti elementi di blocco si posizioni verso il suo lato destro.

```
.picture { float: left; }
```

2. Come influisce la proprietà `top` su un elemento statico rispetto al suo elemento principale?

La proprietà `top` non si applica agli elementi statici.

3. In che modo cambiare la proprietà `display` di un elemento in `flex` influenza il suo posizionamento nel flusso normale?

Il posizionamento dell'elemento stesso non cambia, ma i suoi più immediati elementi figli saranno resi orizzontalmente uno accanto all'altro.

4. Quale caratteristica dei CSS ti permette di usare un set separato di regole a seconda delle dimensioni dello schermo?

Le *Media query* permettono al browser di verificare le dimensioni del *viewport* prima di applicare una regola CSS.



Argomento 034: Programmazione JavaScript



034.1 Esecuzione e Sintassi in JavaScript

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 034.1

Peso

1

Arese di Conoscenza Chiave

- Eseguire JavaScript all'interno di un documento HTML
- Comprendere la sintassi di JavaScript
- Aggiungere commenti al codice JavaScript
- Accedere alla console JavaScript
- Scrivere nella console JavaScript

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- `<script>`, inclusi gli attributi `type` (`text/javascript`) e `src`
- ;
- //, /* */
- `console.log`



034.1 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	034 Programmazione JavaScript
Obiettivo:	034.1 Esecuzione e Sintassi in JavaScript
Lezione:	1 di 1

Introduzione

Le pagine web sono sviluppate utilizzando tre tecnologie standard: HTML, CSS e JavaScript. JavaScript è un linguaggio di programmazione che permette al browser di aggiornare dinamicamente il contenuto di un sito web. Il JavaScript è solitamente eseguito dallo stesso browser usato per visualizzare una pagina web. Questo significa che, come per i CSS e l'HTML, il comportamento esatto di qualsiasi codice che si scrive potrebbe differire tra i vari browser. Fortunatamente i browser più comuni aderiscono alla specifica ECMAScript. Questo è uno standard che unifica l'uso di JavaScript nel web, e sarà la base per la lezione, insieme alla specifica HTML5, che indica come JavaScript deve essere inserito in una pagina web affinché un browser lo esegua.

Esecuzione di JavaScript nel Browser

Per eseguire JavaScript, il browser ha bisogno di ottenere il codice o direttamente, come parte dell'HTML che compone la pagina web, o come una URL che indica una posizione per uno script da eseguire.

L'esempio seguente mostra come includere il codice direttamente nel file HTML:

```

<html>
  <head>
  </head>
  <body>
    <h1>Website Headline</h1>
    <p>Content</p>

    <script>
      console.log('test');
    </script>

  </body>
</html>

```

Il codice è inserito tra i tag `<script>` e `</script>`. Tutto ciò che è incluso in questi tag sarà eseguito direttamente dal browser quando si andrà a caricare la pagina.

La posizione dell'elemento `<script>` all'interno della pagina indica con precisione quando il codice JavaScript verrà eseguito. Un documento HTML viene analizzato dall'alto verso il basso, e il browser decide quando visualizzare gli elementi sullo schermo. Nell'esempio appena mostrato, i tag `<h1>` e `<p>` del sito web vengono analizzati, e probabilmente visualizzati, prima dell'esecuzione dello script. Se il codice JavaScript all'interno del tag `<script>` impiegasse molto tempo per essere eseguito, la pagina verrebbe comunque visualizzata senza alcun problema. Se, tuttavia, lo script fosse stato messo sopra gli altri tag, il visitatore della pagina web dovrebbe aspettare che lo script finisca di essere eseguito prima di vedere la pagina. Per questa ragione, i tag `<script>` sono solitamente posizionati in uno dei due posti:

- Alla fine del “corpo” HTML, in modo che lo script sia l'ultima cosa che viene eseguita. Fatelo quando il codice aggiunge qualcosa alla pagina che non sarebbe utile senza il contenuto. Un esempio potrebbe essere l'aggiunta di funzionalità a un pulsante, dato che il pulsante deve esistere perché la funzionalità abbia senso.
- All'interno dell'elemento `<head>` dell'HTML. Questo assicura che lo script venga eseguito prima che il corpo dell'HTML venga analizzato. Se vuoi cambiare il comportamento di caricamento della pagina, o hai qualcosa che deve essere eseguito mentre la pagina non è ancora completamente caricata, puoi mettere lo script qui. Inoltre, se hai più script che dipendono da un particolare script, puoi mettere quello script condiviso in questa sezione HTML per assicurarti che venga eseguito prima degli altri script.

Per una varietà di ragioni, inclusa la gestibilità, è utile mettere il tuo codice JavaScript in file separati che esistono al di fuori del codice HTML. I file JavaScript esterni sono inclusi usando un tag `<script>` con un attributo `src`, nel modo seguente:

```
<html>
  <head>
    <script src="/button-interaction.js"></script>
  </head>
  <body>
    </body>
</html>
```

Il tag `src` comunica al browser la posizione del sorgente, cioè il file che contiene il codice JavaScript. La posizione può essere un file sullo stesso server, come nell'esempio precedente, o qualsiasi URL accessibile come <https://www.lpi.org/example.js>. Il valore dell'attributo `src` segue la stessa convenzione dell'importazione di file CSS o delle immagini, nel senso che può essere sia relativo sia assoluto. Quando si incontra un tag `script` con l'attributo `src`, il browser cercherà di ottenere il file sorgente usando una richiesta HTTP GET, quindi i file esterni devono essere accessibili.

Quando usi l'attributo `src`, qualsiasi codice o testo posto tra i tag `<script>...</script>` viene ignorato, secondo le specifiche HTML.

```
<html>
  <head>
    <script src="/button-interaction.js">
      console.log("test"); // <-- This is ignored
    </script>
  </head>
  <body>
    </body>
</html>
```

Ci sono altri attributi che puoi aggiungere al tag `script` per specificare ulteriormente come il browser dovrebbe ottenere i file e come dovrebbe gestirli in seguito. La seguente lista entra nel dettaglio sugli attributi più importanti:

async

Può essere usato sui tag `script` e indica al browser di recuperare lo script in background, in modo da non bloccare il processo di caricamento della pagina. Il caricamento della pagina sarà ancora interrotto dopo che il browser avrà ottenuto lo script, perché il browser deve analizzarlo, cosa che viene fatta immediatamente dopo che lo script è stato recuperato completamente. Questo attributo è booleano, quindi scrivere il tag come `<script async src="/script.js"></script>` è sufficiente e non è necessario fornire alcun valore.

defer

Simile a `async`, questo indica al browser di non bloccare il processo di caricamento della pagina mentre recupera lo script. Piuttosto, il browser rinvierà l'analisi dello script. Il browser aspetterà che l'intero documento HTML sia stato analizzato e solo allora analizzerà lo script, prima di annunciare che il documento è stato caricato completamente. Come `async`, `defer` è un attributo booleano ed è usato nello stesso modo. Poiché `defer` implica `async`, non è utile specificare entrambi i tag insieme.

NOTE

Quando una pagina è completamente analizzata, il browser indica che è pronta per la visualizzazione innescando un evento `DOMContentLoaded`, quando il visitatore sarà in grado di vedere il documento. Così, il JavaScript incluso in un evento `<head>` ha sempre la possibilità di agire sulla pagina prima che venga visualizzata, anche se si include l'attributo `defer`.

type

Indica il tipo di script che il browser dovrebbe aspettarsi all'interno del tag. Il default è JavaScript (`type="application/javascript"`), quindi questo attributo non è necessario quando si include codice JavaScript o si punta a una risorsa JavaScript con il tag `src`. Generalmente, tutti i tipi MIME possono essere specificati, ma solo gli script che sono denotati come JavaScript saranno eseguiti dal browser. Ci sono due casi d'uso realistici per questo attributo: dire al browser di non eseguire lo script impostando `type` ad un valore arbitrario come `template` o `other`, o dire al browser che lo script è un modulo ES6. Non tratteremo i moduli ES6 in questa lezione.

WARNING

Quando più script hanno l'attributo `async` impostato, saranno eseguiti nell'ordine in cui finiscono di essere scaricati, *non* nell'ordine dei tag `script` nel documento. L'attributo `defer`, invece, mantiene l'ordine dei tag `script`.

La Console del Browser

Mentre di solito viene eseguito come parte di un sito web, c'è un altro modo di eseguire JavaScript: attraverso la console del browser. Tutti i moderni browser desktop forniscono un menu attraverso il quale è possibile eseguire codice JavaScript nel motore JavaScript del browser. Questo di solito viene fatto per testare nuovo codice o per eseguire il debug di siti web esistenti.

Ci sono diversi modi per accedere alla console del browser. Il modo più semplice è attraverso le scorciatoie da tastiera. Ecco di seguito le scorciatoie da tastiera per alcuni dei browser attualmente più in uso:

Chrome

`Ctrl + Shift + J` (`Cmd + Option + J` sul Mac)

Firefox

`Ctrl + Shift + K` (`Cmd + Option + K` sul Mac)

Safari

`Ctrl + Shift + ?` (`Cmd + Option + ?` sul Mac)

Puoi anche cliccare con il tasto destro del mouse su una pagina web e selezionare l'opzione “Ispeziona” o “Ispeziona Elemento”. Nel pannello che verrà ad aprirsi, puoi selezionare la scheda “Console”; ciò farà apparire la console del browser.

Una volta richiamata la console, è possibile eseguire JavaScript sulla pagina digitando il codice direttamente nel campo di input. Il risultato di qualsiasi codice eseguito sarà mostrato in una riga separata.

Istruzioni in JavaScript

Ora che sappiamo come avviare l'esecuzione di uno script, copriremo le basi di come uno script viene effettivamente eseguito. Uno script JavaScript è un insieme di istruzioni e blocchi. Un esempio di istruzione è `console.log('test')`. Questa istruzione dice al browser di scrivere la parola `test` nella console del browser.

Ogni istruzione (statement) in JavaScript è terminata da un punto e virgola (`;`). Questo dice al browser che l'istruzione è terminata e che una nuova può essere iniziata. Considerate il seguente script:

```
var message = "test"; console.log(message);
```

Abbiamo scritto due istruzioni. Ciascuna è terminata da un punto e virgola o dalla fine dello script. Per motivi di leggibilità, possiamo mettere le istruzioni su linee separate. In questo modo, lo script potrebbe anche essere scritto come:

```
var message = "test";
console.log(message);
```

Questo è possibile perché tutti gli spazi bianchi tra le istruzioni (come uno spazio, una nuova linea o un tab) vengono ignorati. Gli spazi bianchi possono anche essere spesso messi tra le singole parole

chiave all'interno delle istruzioni, ma questo sarà ulteriormente spiegato in una prossima lezione. Le istruzioni possono anche essere vuote o composte solo da spazi bianchi.

Se un'istruzione non è valida perché non è stata terminata da un punto e virgola, ECMAScript farà un tentativo per inserire automaticamente il punto e virgola corretto, sulla base di un complesso insieme di regole. La regola più importante è: Se un'istruzione non valida è composta da due istruzioni valide separate da una linea nuova, inserire un punto e virgola alla linea nuova. Per esempio, il seguente codice non forma un'istruzione valida:

```
console.log("hello")
console.log("wor ld")
```

Un browser moderno però lo eseguirà automaticamente come se fosse scritto con il punto e virgola appropriato:

```
console.log("hello");
console.log("wor ld");
```

Pertanto, è possibile omettere il punto e virgola in alcuni casi. Tuttavia, poiché le regole per l'inserimento automatico del punto e virgola sono complesse, vi consigliamo di terminare sempre correttamente le vostre istruzioni per evitare errori indesiderati.

I Commenti in JavaScript

Script estesi possono diventare abbastanza complicati. Potresti voler commentare ciò che stai scrivendo, per rendere lo script più facile da leggere ad altre persone, o per te stesso in futuro. In alternativa, potresti voler includere *meta informazioni* nello script, come informazioni sul copyright, o informazioni su quando lo script è stato scritto e perché.

Per rendere possibile l'inclusione di tali meta informazioni, JavaScript supporta i *commenti*. Uno sviluppatore può includere caratteri speciali in uno script che denotano certe parti di esso come un commento, che sarà saltato durante l'esecuzione. Quella che segue è una versione commentata dello script che abbiamo visto precedentemente.

```
/*
This script was written by the author of this lesson in May, 2020.
It has exactly the same effect as the previous script, but includes comments.
*/
// First, we define a message.
var message = "test";
console.log(message); // Then, we output the message to the console.
```

I commenti “non” sono istruzioni e non hanno bisogno di essere terminati da un punto e virgola. Seguono invece le proprie regole per la terminazione, a seconda del modo in cui il commento è stato scritto. Ci sono due modi per scrivere commenti in JavaScript:

Commento multi-linea

Usa `/*` e `*/` per segnalare l'inizio e la fine di un commento multilinea. Tutto ciò che viene dopo `/*`, fino alla prima occorrenza di `*/` viene ignorato. Questo tipo di commento è generalmente usato per coprire linee multiple, ma può anche essere usato per linee singole, o anche all'interno di una linea come la seguente:

```
console.log(/* what we want to log: */ "hello world")
```

Poiché l'obiettivo dei commenti è generalmente quello di aumentare la leggibilità di uno script, si dovrebbe evitare di usare questo stile di commento all'interno di una linea.

Commento su singola linea

Usa `//` (doppio slash) per *commentare* una linea. Tutto ciò che viene dopo, sulla stessa linea, viene ignorato. Nell'esempio mostrato prima, questo schema è usato prima per commentare un'intera linea. Dopo l'istruzione `console.log(message);`, si usa per scrivere un commento sul resto della linea.

In linea generale e per semplicità, i commenti a linea singola dovrebbero essere usati per linee singole, e i commenti a più linee per linee multiple. I commenti all'interno di un'istruzione dovrebbero essere evitati.

I commenti possono anche essere usati per rimuovere temporaneamente linee di codice attivo, come nel modo seguente:

```
// We temporarily want to use a different message
// var message = "test";
var message = "something else";
```

Esercizi Guidati

1. Crea una variabile chiamata ColorName e assegnamele il valore RED.

2. Quali dei seguenti script sono validi?

console.log("hello") console.log("world");	
console.log("hello"); console.log("world");	
// console.log("hello") console.log("world");	
console.log("hello"); console.log("world") //;	
console.log("hello"); /* console.log("world") */	

Esercizi Esplorativi

1. Quante istruzioni JavaScript possono essere scritte su una singola linea senza usare un punto e virgola?

2. Crea due variabili chiamate `x` e `y`, poi mostra la loro somma nella console.

Sommario

In questa lezione abbiamo imparato diversi modi di eseguire JavaScript e come modificare il comportamento del caricamento degli script. Abbiamo anche imparato i concetti base della composizione e del commento degli script, e abbiamo imparato a usare il comando `console.log()`.

HTML utilizzato in questa lezione:

`<script>`

Il tag `script` può essere usato per includere JavaScript direttamente o specificando un file con l'attributo `src`. Modificare come lo script viene caricato con gli attributi `async` e `defer`.

Concetti JavaScript introdotti in questa lezione:

;

Il punto e virgola è usato per separare le istruzioni. I punti e virgola possono - ma non dovrebbero - essere omessi.

`//, /*...*/`

I commenti possono essere usati per aggiungere commenti o meta informazioni a un file di script, o per evitare che le istruzioni vengano eseguite.

`console.log("text")`

Il comando `console.log()` può essere usato per scrivere testo nella console del browser.

Risposte agli Esercizi Guidati

1. Crea una variabile chiamata ColorName e assegnamele il valore RED.

```
var ColorName = "RED";
```

2. Quali dei seguenti script sono validi?

console.log("hello") console.log("world");	Non valido: Il primo comando console.log() non è terminato correttamente e l'intera linea non forma un'istruzione valida.
console.log("hello"); console.log("world");	Valido: Ogni istruzione è terminata correttamente.
// console.log("hello") console.log("world");	Valido: L'intero codice viene ignorato perché è un commento.
console.log("hello"); console.log("world") //;	Non valido: All'ultima istruzione manca un punto e virgola. Il punto e virgola alla fine viene ignorato perché è commentato.
console.log("hello"); /* console.log("world") */	Valido: Un'istruzione valida è seguita da codice commentato, che viene ignorato.

Risposte agli Esercizi Esplorativi

1. Quante istruzioni JavaScript possono essere scritte su una singola linea senza usare un punto e virgola?

Se siamo alla fine di uno script, possiamo scrivere un'istruzione e questa sarà terminata dalla fine del file. Altrimenti, non è possibile scrivere un'istruzione senza punto e virgola con la sintassi che hai imparato finora.

2. Crea due variabili chiamate `x` e `y`, poi mostra la loro somma nella console.

```
var x = 5;  
var y = 10;  
console.log(x+y);
```



034.2 Strutture di Dati in JavaScript

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 034.2

Peso

3

Arese di Conoscenza Chiave

- Definire e usare variabili e costanti
- Comprendere i tipi di dati
- Comprendere la conversione/coerenza dei tipi
- Comprendere gli array e gli oggetti
- Conoscenza dell’ambito delle variabili

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- `=, +, -, *, /, %, --, ++, +=, -=, *=, /=`
- `var, let, const`
- `boolean, number, string, symbol`
- `array, object`
- `undefined, null, NaN`



034.2 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	034 Programmazione JavaScript
Obiettivo:	034.2 Strutture di Dati in JavaScript
Lezione:	1 di 1

Introduzione

I linguaggi di programmazione, come i linguaggi naturali, rappresentano la realtà attraverso simboli che sono combinati in dichiarazioni significative. La realtà rappresentata da un linguaggio di programmazione sono le risorse della macchina, come le operazioni del processore, i dispositivi e la memoria.

Ognuno della miriade di linguaggi di programmazione adotta un paradigma per rappresentare le informazioni. JavaScript adotta convenzioni tipiche dei linguaggi di *alto livello*, dove la maggior parte dei dettagli come l'allocazione della memoria sono impliciti, permettendo al programmatore di concentrarsi sullo scopo dello script nel contesto dell'applicazione.

Linguaggi di Alto Livello

I linguaggi di alto livello forniscono regole astratte in modo che il programmatore abbia bisogno di scrivere meno codice per esprimere un'idea. JavaScript offre modi convenienti per fare uso della memoria del computer, utilizzando concetti di programmazione che semplificano la scrittura di pratiche ricorrenti e che sono generalmente sufficienti per lo scopo dello sviluppatore web.

NOTE

Anche se è possibile utilizzare meccanismi specializzati per un accesso meticoloso alla memoria, i tipi di dati più semplici che vedremo sono di uso più generale.

Le operazioni tipiche in un'applicazione web consistono nel richiedere dati attraverso qualche istruzione JavaScript e nel memorizzarli per essere elaborati ed eventualmente presentati all'utente. Questa memorizzazione è abbastanza flessibile in JavaScript, con formati di memorizzazione adatti a ogni scopo.

Dichiarazione di Costanti e Variabili

La dichiarazione di costanti e variabili per contenere dati è la pietra angolare di qualsiasi linguaggio di programmazione. JavaScript adotta la convenzione della maggior parte dei linguaggi di programmazione, assegnando valori a costanti o variabili con la sintassi `nome = valore`. La costante o la variabile a sinistra prende il valore a destra. Il nome della costante o della variabile deve iniziare con una lettera o un *underscore*.

Il tipo di dati memorizzati nella variabile non ha bisogno di essere indicato, perché JavaScript è un linguaggio a *tipizzazione dinamica*. Il tipo della variabile viene dedotto dal valore che le viene assegnato. Tuttavia, è conveniente designare certi attributi nella dichiarazione per garantire il risultato atteso.

NOTE

TypeScript è un linguaggio ispirato a JavaScript che, come i linguaggi *low_level*, permette di dichiarare variabili per specifici tipi di dati.

Costanti

Una costante è un simbolo che viene assegnato *solo una volta* all'avvio del programma e non cambia mai. Le costanti sono utili per specificare valori fissi come definire la costante PI con 3.14159265, o COMPANY_NAME con il nome della tua azienda.

In un'applicazione web, per esempio, prendiamo un client che riceve informazioni meteorologiche da un server remoto. Il programmatore può decidere che l'indirizzo del server debba essere costante, perché non cambierà durante l'esecuzione dell'applicazione. Le informazioni sulla temperatura, tuttavia, possono cambiare a ogni nuovo arrivo di dati dal server.

L'intervallo tra le interrogazioni fatte al server può anche essere definito come una costante, che può essere interrogata da qualsiasi parte del programma:

```
const update_interval = 10;

function setup_app(){
    console.log("Update every " + update_interval + "minutes");
}
```

Quando viene invocata, la funzione `setup_app()` visualizza il messaggio `Update every 10 minutes` sulla console. Il termine `const` posto prima del nome `update_interval` fa sì che il suo valore rimanga lo stesso durante l'intera esecuzione dello script. Se si tenta di reimpostare il valore di una costante, si verifica un errore: `TypeError: Assignment to constant variable`.

Variabili

Senza il termine `const`, JavaScript assume automaticamente che `update_interval` sia una variabile e che il suo valore possa essere modificato. Questo equivale a dichiarare esplicitamente la variabile con `var`:

```
var update_interval;
update_interval = 10;

function setup_app(){
    console.log("Update every " + update_interval + "minutes");
}
```

Sebbene la variabile `update_interval` sia stata definita al di fuori della funzione, vi si accede dall'interno della funzione. Qualsiasi costante o variabile dichiarata al di fuori di funzioni o blocchi di codice definiti da parentesi graffe (`{}`) ha uno *scopo globale*; cioè, vi si può accedere da qualsiasi parte del codice. Attenzione perché non è vero il contrario: una costante o una variabile dichiarata all'interno di una funzione ha uno *scopo locale*, quindi è accessibile solo dall'interno della funzione stessa. I blocchi di codice delimitati da parentesi, come quelli inseriti nelle strutture decisionali `if` o nei cicli `for`, delimitano lo scopo delle costanti, ma non delle variabili dichiarate come `var`. Il seguente codice, per esempio, è valido:

```
var success = true;
if ( success == true )
{
    var message = "Transaction succeeded";
    var retry = 0;
}
else
{
    var message = "Transaction failed";
    var retry = 1;
}

console.log(message);
```

L'istruzione `console.log(message)` è in grado di accedere alla variabile `message`, anche se è stata dichiarata all'interno del blocco di codice della struttura `if`. Lo stesso non accadrebbe se `message` fosse costante, come esemplificato nell'esempio seguente:

```
var success = true;
if ( success == true )
{
    const message = "Transaction succeeded";
    var retry = 0;
}
else
{
    const message = "Transaction failed";
    var retry = 1;
}

console.log(message);
```

In questo caso, verrebbe emesso un messaggio di errore del tipo `ReferenceError: message is not defined` e lo script verrebbe fermato. Anche se può sembrare una limitazione, restringere lo scopo delle variabili e delle costanti aiuta a evitare la confusione tra le informazioni trattate nel corpo dello script e suoi diversi blocchi di codice. Per questo motivo, le variabili dichiarate con `let` invece di `var` sono anche limitate nello scopo dai blocchi delimitati da parentesi graffe. Ci sono altre sottili differenze tra dichiarare una variabile con `var` o con `let`, ma la più significativa riguarda lo “scopo” della variabile, come discusso qui di seguito.

Tipi di Valore

La maggior parte delle volte il programmatore non ha bisogno di preoccuparsi del tipo di dati memorizzati in una variabile, perché JavaScript li identifica automaticamente con uno dei suoi tipi *primitivi* durante la prima assegnazione di un valore alla variabile. Alcune operazioni, tuttavia, possono essere specifiche per un tipo di dati o un altro e possono provocare errori se usate senza discrezione. Inoltre, JavaScript offre tipi *strutturati* che permettono di combinare più di un tipo primitivo in un singolo oggetto.

Tipi Primitivi

Le istanze di tipo primitivo corrispondono alle variabili tradizionali, che memorizzano un solo valore. I tipi sono definiti implicitamente, quindi l'operatore `typeof` può essere usato per identificare quale tipo di valore è memorizzato in una variabile:

```
console.log("Undefined variables are of type", typeof variable);

{
  let variable = true;
  console.log("Value 'true' is of type " + typeof variable);
}

{
  let variable = 3.14159265;
  console.log("Value '3.14159265' is of type " + typeof variable);
}

{
  let variable = "Text content";
  console.log("Value 'Text content' is of type " + typeof variable);
}

{
  let variable = Symbol();
  console.log("A symbol is of type " + typeof variable);
}
```

Questo script mostrerà sulla console che tipo di variabile è stata usata in ognuno dei casi:

```
undefined variables are of type undefined
Value 'true' is of type boolean
Value '3.114159265' is of type number
Value 'Text content' is of type string
A symbol is of type symbol
```

Nota che la prima linea cerca di trovare il tipo di una variabile non dichiarata. Questo fa sì che la variabile data sia identificata come `undefined`. Il tipo `symbol` è la primitiva meno intuitiva. Il suo scopo è quello di fornire un nome di attributo unico all'interno di un oggetto quando non c'è bisogno di definire un nome di attributo specifico. Un oggetto è una delle strutture dati che vedremo in seguito.

Tipi Strutturati

Mentre i tipi primitivi sono sufficienti per scrivere semplici routine, ci sono svantaggi nell'usarli esclusivamente in applicazioni più complesse. Un'applicazione di e-commerce, per esempio, sarebbe molto più difficile da scrivere, perché il programmatore dovrebbe trovare il modo di memorizzare liste di articoli e valori corrispondenti usando solo variabili con tipi primitivi.

I tipi strutturati semplificano il compito di raggruppare informazioni della stessa natura in una singola variabile. Una lista di articoli in un carrello della spesa, per esempio, può essere memorizzata in una singola variabile di tipo `array`:

```
let cart = ['Milk', 'Bread', 'Eggs'];
```

Come dimostrato nell'esempio, un array di elementi è contraddistinto da parentesi quadre. L'esempio ha popolato l'array con tre valori letterali di stringa, da cui l'uso delle virgolette singole. Anche le variabili possono essere usate come elementi di un array, ma in questo caso devono essere designate senza virgolette. Il numero di elementi in un array può essere interrogato con la proprietà `length`:

```
let cart = ['Milk', 'Bread', 'Eggs'];
console.log(cart.length);
```

Il numero `3` sarà visualizzato nell'output della console. Nuovi elementi possono essere aggiunti all'array con il metodo `push()`:

```
cart.push('Candy');
console.log(cart.length);
```

Questa volta, la cifra visualizzata sarà 4. Ogni elemento della lista può essere raggiunto tramite il suo indice numerico, a partire da 0:

```
console.log(cart[0]);
console.log(cart[3]);
```

L'output visualizzato sulla console sarà:

```
Milk
Candy
```

Proprio come è possibile usare `push()` per aggiungere un elemento, è anche possibile usare `pop()` per rimuovere l'ultimo elemento da un array.

Non è necessario che i valori memorizzati in una matrice siano dello stesso tipo. È possibile, per esempio, memorizzare la quantità di ogni articolo accanto a esso. Una lista della spesa come quella dell'esempio precedente potrebbe essere costruita come segue:

```
let cart = ['Milk', 1, 'Bread', 4, 'Eggs', 12, 'Candy', 2];

// Item indexes are even
let item = 2;

// Quantities indexes are odd
let quantity = 3;

console.log("Item: " + cart[item]);
console.log("Quantity: " + cart[quantity]);
```

L'output visualizzato sulla console dopo l'esecuzione di questo codice è:

```
Item: Bread
Quantity: 4
```

Come avrai già notato, combinare i nomi degli elementi con le loro rispettive quantità in un unico

array può non essere una buona idea, perché la relazione tra loro non è esplicita nella struttura dei dati ed è molto suscettibile di errori (umani). Anche se si usasse un array per i nomi e un altro array per le quantità, mantenere l'integrità della lista richiederebbe la stessa cura e non sarebbe molto produttivo. In queste situazioni, l'alternativa migliore è usare una struttura di dati più appropriata: un *oggetto*.

In JavaScript una struttura di dati di tipo oggetto permette di legare proprietà a una variabile. Inoltre, a differenza di un array, gli elementi che compongono un oggetto non hanno un ordine fisso. Un elemento della lista della spesa, per esempio, può essere un oggetto con le proprietà `name` e `quantity`:

```
let item = { name: 'Milk', quantity: 1 };
console.log("Item: " + item.name);
console.log("Quantity: " + item.quantity);
```

Questo esempio mostra che un oggetto può essere definito usando le parentesi graffe (`{}`), dove ogni coppia proprietà/valore è separata da due punti e le proprietà sono separate da virgole. La proprietà è accessibile nel formato `variable.property`, come in `item.name`, sia per la lettura che per l'assegnazione di nuovi valori. L'output visualizzato sulla console dopo l'esecuzione di questo codice è:

```
Item: Milk
Quantity: 1
```

Infine, ogni oggetto che rappresenta un elemento può essere incluso nell'array della lista della spesa. Questo può essere fatto direttamente quando si crea la lista:

```
let cart = [{ name: 'Milk', quantity: 1 }, { name: 'Bread', quantity: 4 }];
```

Come prima, un nuovo oggetto che rappresenta un elemento può essere aggiunto successivamente all'array:

```
cart.push({ name: 'Eggs', quantity: 12 });
```

Gli elementi della lista sono ora accessibili tramite il loro indice e il loro nome di proprietà:

```
console.log("Third item: " + cart[2].name);
console.log(cart[2].name + " quantity: " + cart[2].quantity);
```

L'output visualizzato sulla console dopo l'esecuzione di questo codice è:

```
third item: eggs  
Eggs quantity: 12
```

Le strutture di dati permettono al programmatore di mantenere il proprio codice molto più organizzato e più facile da mantenere, sia per l'autore originale che per altri programmatori del team. Inoltre, molti output delle funzioni JavaScript sono in tipi strutturati, che devono essere gestiti correttamente dal programmatore.

Operatori

Finora abbiamo praticamente visto solo come assegnare valori alle variabili appena create. Per quanto semplice, qualsiasi programma eseguirà diverse altre manipolazioni sui valori delle variabili. JavaScript offre diversi tipi di *operatori* che possono agire direttamente sul valore di una variabile o memorizzare il risultato dell'operazione in una nuova variabile.

La maggior parte degli operatori è orientata alle operazioni aritmetiche. Per aumentare la quantità di un elemento nella lista della spesa, per esempio, basta usare l'operatore di addizione +:

```
item.quantity = item.quantity + 1;
```

Il seguente codice mostra il valore di `item.quantity` prima e dopo l'aggiunta. Non confondete i ruoli del segno più nello codice. Le istruzioni di `console.log` usano il segno più per combinare due stringhe.

```
let item = { name: 'Milk', quantity: 1 };  
console.log("Item: " + item.name);  
console.log("Quantity: " + item.quantity);  
  
item.quantity = item.quantity + 1;  
console.log("New quantity: " + item.quantity);
```

L'output visualizzato sulla console dopo aver eseguito questo codice è:

```
Item: Milk  
Quantity: 1  
New quantity: 2
```

Si noti che il valore precedentemente memorizzato in `item.quantity` viene utilizzato come operando dell'operazione di addizione: `item.quantity = item.quantity + 1`. Solo al termine dell'operazione il valore in `item.quantity` viene aggiornato con il risultato dell'operazione. Questo tipo di operazione aritmetica che coinvolge il valore corrente della variabile di destinazione è abbastanza comune, quindi esistono operatori abbreviati che consentono di scrivere la stessa operazione in un formato ridotto:

```
item.quantity += 1;
```

Anche le altre operazioni di base hanno operatori stenografici equivalenti:

- `a = a - b` è equivalente a `a -= b`.
- `a = a * b` è equivalente a `a *= b`.
- `a = a / b` è equivalente a `a /= b`.

Per l'addizione e la sottrazione, è disponibile un terzo formato quando il secondo operando è solo un'unità:

- `a = a + 1` è equivalente a `a++`.
- `a = a - 1` è equivalente a `a--`.

È possibile combinare più operatori nella stessa operazione e il risultato può essere memorizzato in una nuova variabile. Per esempio, la seguente istruzione calcola il prezzo totale di un articolo più un costo di spedizione:

```
let total = item.quantity * 9.99 + 3.15;
```

L'ordine in cui vengono eseguite le operazioni segue il tradizionale ordine di precedenza: prima vengono eseguite le operazioni di moltiplicazione e divisione, e solo dopo vengono eseguite le operazioni di addizione e sottrazione. Gli operatori con la stessa precedenza vengono eseguiti nell'ordine in cui compaiono nell'espressione, da sinistra a destra. Per sovrascrivere l'ordine di precedenza predefinito, puoi utilizzare le parentesi, come in `a * (b + c)`. In alcune situazioni, il risultato di un'operazione non ha nemmeno bisogno di essere memorizzato in una variabile. Questo è il caso in cui si desidera valutare il risultato di un'espressione all'interno di un'istruzione `if`:

```
if ( item.quantity % 2 == 0 )
{
    console.log("Quantity for the item is even");
}
else
{
    console.log("Quantity for the item is odd");
}
```

L'operatore `%` (modulo) restituisce il resto della divisione del primo operando per il secondo operando. Nell'esempio, l'istruzione `if` verifica se il resto della divisione di `item.quantity` per 2 è uguale a zero, ovvero se `item.quantity` è un multiplo di 2.

Quando uno degli operandi dell'operatore `+` è una stringa, gli altri operatori sono *concatenati* in stringhe e il risultato è una concatenazione di stringhe. Negli esempi precedenti, questo tipo di operazione è stato usato per concatenare stringhe e variabili nell'argomento dell'istruzione `console.log`.

Questa conversione automatica potrebbe non essere il comportamento desiderato. Un valore fornito dall'utente in un campo del modulo, per esempio, potrebbe essere identificato come una stringa, ma in realtà è un valore numerico. In casi come questo, la variabile deve prima essere convertita in un numero con la funzione `Number()`:

```
sum = Number(value1) + value2;
```

Inoltre, è importante verificare che l'utente abbia fornito un valore valido prima di procedere con l'operazione. In JavaScript, una variabile senza un valore assegnato contiene il valore `null`. Questo permette al programmatore di usare una dichiarazione di decisione come `if (value1 == null)` per controllare se a una variabile è stato assegnato un valore, indipendentemente dal tipo di valore assegnato alla variabile.

Esercizi Guidati

1. Un array è una struttura di dati presente in diversi linguaggi di programmazione, alcuni dei quali permettono solo array con elementi dello stesso tipo. Nel caso di JavaScript, è possibile definire un array con elementi di tipo diverso?

2. Sulla base dell'esempio `let item = { name: 'Milk', quantity: 1 }` per un oggetto in una lista della spesa, come potrebbe essere dichiarato questo oggetto per includere il prezzo dell'articolo?

3. In una sola riga di codice, quali sono i modi per aggiornare il valore di una variabile alla metà del suo valore attuale?

Esercizi Esplorativi

- Nel seguente codice, quale valore verrà visualizzato nell'output della console?

```
var value = "Global";  
  
{  
    value = "Location";  
}  
  
console.log(value);
```

- Che cosa succede quando uno o più degli operandi coinvolti in un'operazione di moltiplicazione è una stringa?

- Come è possibile rimuovere l'elemento Eggs dall'array cart dichiarato con `let cart = ['Milk', 'Bread', 'Eggs']`?

Sommario

Questa lezione spiega l'uso di base delle costanti e delle variabili in JavaScript. JavaScript è un *linguaggio a tipizzazione dinamica*, quindi il programmatore non ha bisogno di specificare il tipo di variabile prima di impostarla. Tuttavia, è importante conoscere i tipi primitivi del linguaggio per assicurare il corretto risultato delle operazioni di base. Inoltre, le strutture dati come gli array e gli oggetti combinano i tipi primitivi e permettono al programmatore di costruire variabili più complesse e composite. Questa lezione passa in rassegna i seguenti concetti e procedure:

- Capire le costanti e le variabili
- Ambito delle variabili
- Dichiарare le variabili con `var` e `let`
- Tipi primitivi
- Operatori aritmetici
- Array e oggetti
- Coercizione e conversione dei tipi

Risposte agli Esercizi Guidati

1. Un array è una struttura di dati presente in diversi linguaggi di programmazione, alcuni dei quali permettono solo array con elementi dello stesso tipo. Nel caso di JavaScript, è possibile definire un array con elementi di tipo diverso?

Sì, in JavaScript è possibile definire array con elementi di diversi tipi primitivi, come stringhe e numeri.

2. Sulla base dell'esempio `let item = { name: 'Milk', quantity: 1 }` per un oggetto in una lista della spesa, come potrebbe essere dichiarato questo oggetto per includere il prezzo dell'articolo?

```
let item = { name: 'Milk', quantity: 1, price: 4.99 };
```

3. In una sola riga di codice, quali sono i modi per aggiornare il valore di una variabile alla metà del suo valore attuale?

Si può usare la variabile stessa come operando, `value = value / 2`, o l'operatore abbreviato `/=`: `value /= 2`.

Risposte agli Esercizi Esplorativi

- Nel seguente codice, quale valore verrà visualizzato nell'output della console?

```
var value = "Global";  
  
{  
    value = "Location";  
}  
  
console.log(value);
```

Location

- Che cosa succede quando uno o più degli operandi coinvolti in un'operazione di moltiplicazione è una stringa?

JavaScript assegnerà il valore NaN (Not a Number) al risultato, indicando che l'operazione non è valida.

- Come è possibile rimuovere l'elemento Eggs dall'array cart dichiarato con `let cart = ['Milk', 'Bread', 'Eggs']`?

Gli array in JavaScript hanno il metodo `pop()`, che rimuove l'ultimo elemento della lista: `cart.pop()`.



034.3 Strutture di Controllo e Funzioni in JavaScript

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 034.3

Peso

4

Arese di Conoscenza Chiave

- Comprendere i valori veri e falsi
- Comprendere gli operatori di confronto
- Comprendere la differenza tra confronto libero e rigido
- Usare i condizionali
- Usare i cicli
- Definire funzioni personalizzate

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- `if, else if, else`
- `switch, case, break`
- `for, while, break, continue`
- `function, return`
- `==, !=, <, >, >=`
- `====, !===`



034.3 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	034 Programmazione JavaScript
Obiettivo:	034.3 Strutture di Controllo e Funzioni in JavaScript
Lezione:	1 di 2

Introduzione

Come qualsiasi altro linguaggio di programmazione il codice JavaScript è un insieme di dichiarazioni che dicono a un interprete di istruzioni che cosa fare in ordine sequenziale. Tuttavia, questo non significa che ogni istruzione debba essere eseguita solo una volta o che debba essere eseguita affatto. La maggior parte delle istruzioni dovrebbe essere eseguita solo quando si verificano condizioni specifiche. Anche quando uno script è attivato in modo asincrono da eventi indipendenti, spesso deve controllare un certo numero di variabili di controllo per trovare la giusta porzione di codice da eseguire.

Dichiarazioni If

La struttura di controllo più semplice è data dall'istruzione `if`, che eseguirà l'istruzione immediatamente successiva se la condizione specificata è vera. JavaScript considera le condizioni *vere* se il valore valutato è diverso da zero. Qualsiasi cosa all'interno della parentesi dopo la parola `if` (gli spazi sono ignorati) sarà interpretata come una condizione. Nell'esempio seguente, il numero letterale `1` è la condizione:

```
if ( 1 ) console.log("1 is always true");
```

Il numero `1` è scritto esplicitamente in questa condizione di esempio, quindi è trattato come un valore costante (rimane lo stesso durante l'esecuzione dello script) e darà sempre come risultato vero quando viene usato come espressione condizionale. La parola `true` (senza le doppie virgolette) potrebbe anche essere usata al posto di `1`, dato che è anche trattata come un valore letterale vero dal linguaggio. L'istruzione `console.log` mostra i suoi argomenti nella *console* del browser.

TIP

La console del browser mostra errori, avvisi e messaggi informativi inviati con l'istruzione JavaScript `console.log`. Su Chrome, la combinazione di tasti `Ctrl + Shift + J` (`Cmd + Option + J` su Mac) apre la console. Su Firefox, la combinazione di tasti `Ctrl + Shift + K` (`Cmd + Option + K` su Mac) apre la scheda console negli strumenti di sviluppo.

Anche se sintatticamente corretto, usare espressioni costanti nelle condizioni non è molto utile. In un'applicazione reale, probabilmente vorresti testare la veridicità di una variabile:

```
let my_number = 3;
if ( my_number ) console.log("The value of my_number is", my_number, "and it yields
true");
```

Il valore assegnato alla variabile `my_number` (`3`) non è zero, quindi produce `true`. Ma questo esempio non è di uso comune, perché raramente si ha bisogno di verificare se un numero è uguale a zero. È molto più comune confrontare un valore con un altro e verificare se il risultato è vero:

```
let my_number = 3;
if ( my_number == 3 ) console.log("The value of my_number is", my_number, "indeed
");
```

L'operatore di confronto *doppio-uguale* (`==`) viene utilizzato perché l'operatore uguale (`=`) è già definito come operatore di assegnazione. Il valore su ciascun lato dell'operatore è chiamato *operando*. L'ordine degli operandi non ha importanza e qualsiasi espressione che restituisce un valore può essere un operando. Ecco un elenco di altri operatori di confronto disponibili:

value1 == value2

Vero se `value1` è uguale a `value2`.

value1 != value2

Vero se `value1` non è uguale a `value2`.

value1 < value2

Vero se `value1` è minore di `value2`.

value1 > value2

Vero se `value1` è maggiore di `value2`.

value1 <= value2

Vero se `value1` è minore o uguale a `value2`.

value1 >= value2

Vero se `value1` è maggiore o uguale a `value2`.

Di solito, non importa se l'operando a sinistra dell'operatore è una stringa e l'operando a destra è un numero, purché JavaScript sia in grado di convertire l'espressione in un confronto significativo. Quindi la stringa contenente il carattere `1` sarà trattata come il numero `1` rispetto a una variabile numerica. Per assicurarsi che l'espressione restituisca `true` solo se entrambi gli operandi sono dello stesso tipo e valore, è necessario utilizzare l'operatore di identità rigorosa `==` invece di `=`. Allo stesso modo, l'operatore stretto di non identità `!=` viene valutato come `true` se il primo operando non è dello stesso tipo e valore del secondo operatore.

Opzionalmente, la struttura di controllo `if` può eseguire un'istruzione alternativa quando l'espressione viene valutata come falsa:

```
let my_number = 4;
if ( my_number == 3 ) console.log("The value of my_number is 3");
else console.log("The value of my_number is not 3");
```

L'istruzione `else` dovrebbe seguire immediatamente l'istruzione `if`. Finora stiamo eseguendo solo un'istruzione quando la condizione è soddisfatta. Per eseguire più di un'istruzione, devi racchiuderle tra parentesi graffe:

```
let my_number = 4;
if ( my_number == 3 )
{
    console.log("The value of my_number is 3");
    console.log("and this is the second statement in the block");
}
else
{
    console.log("The value of my_number is not 3");
    console.log("and this is the second statement in the block");
}
```

Un gruppo di una o più istruzioni delimitate da una coppia di parentesi graffe è noto come dichiarazione di blocco. È comune usare dichiarazioni di blocco anche quando c'è una sola istruzione da eseguire, per rendere lo stile di codifica coerente in tutto lo script. Inoltre, JavaScript non richiede che le parentesi graffe o qualsiasi dichiarazione siano su linee separate, ma farlo migliora la leggibilità e rende più facile la manutenzione del codice.

Le strutture di controllo possono essere annidate l'una dentro l'altra, ma è importante non confondere le parentesi graffe di apertura e di chiusura di ogni dichiarazione di blocco:

```
let my_number = 4;

if ( my_number > 0 )
{
    console.log("The value of my_number is positive");

    if ( my_number % 2 == 0 )
    {
        console.log("and it is an even number");
    }
    else
    {
        console.log("and it is an odd number");
    }
} // end of if ( my_number > 0 )
else
{
    console.log("The value of my_number is less than or equal to 0");
    console.log("and I decided to ignore it");
}
```

Le espressioni valutate dall'istruzione `if` possono essere più elaborate di semplici confronti. Questo è il caso dell'esempio precedente, in cui l'espressione aritmetica `my_number % 2` è stata impiegata all'interno delle parentesi nell'istruzione nidificata `if`. L'operatore `%` restituisce il resto dopo aver diviso il numero alla sua sinistra per il numero alla sua destra. Gli operatori aritmetici come `%` hanno la precedenza sugli operatori di confronto come `==`, quindi il confronto utilizzerà il risultato dell'espressione aritmetica come operando sinistro.

In molte situazioni le strutture condizionali annidate possono essere combinate in una singola struttura usando gli operatori logici. Se fossimo interessati solo ai numeri pari positivi, per esempio, si potrebbe usare una singola struttura `if`:

```
let my_number = 4;

if ( my_number > 0 && my_number % 2 == 0 )
{
    console.log("The value of my_number is positive");
    console.log("and it is an even number");
}
else
{
    console.log("The value of my_number either 0, negative");
    console.log("or it is a negative number");
}
```

Il doppio operatore `&&` (*ampersand*) nell'espressione valutata è l'operatore logico *AND*. Valuta come vero solo se l'espressione alla sua sinistra e l'espressione alla sua destra sono vere. Se vuoi abbinare numeri che siano positivi o pari, dovresti invece usare l'operatore `||`, che sta per l'operatore logico *OR*:

```
let my_number = -4;

if ( my_number > 0 || my_number % 2 == 0 )
{
    console.log("The value of my_number is positive");
    console.log("or it is a even negative number");
}
```

In questo esempio, solo i numeri dispari negativi non corrisponderanno ai criteri imposti dall'espressione composta. Se avete l'intento opposto, cioè di far corrispondere solo i numeri dispari negativi, aggiungi l'operatore logico `NOT !` all'inizio dell'espressione:

```
let my_number = -5;

if ( !( my_number > 0 || my_number % 2 == 0 ) )
{
    console.log("The value of my_number is an odd negative number");
}
```

L'aggiunta delle parentesi nell'espressione composta forza l'espressione racchiusa da esse a essere valutata per prima. Senza queste parentesi l'operatore *NOT* si applicherebbe solo a `my_number > 0` e poi verrebbe valutata l'espressione *OR*. Gli operatori `&&` e `||` sono conosciuti come operatori logici *binari*, perché richiedono due operandi. `!` è conosciuto come operatore logico *unario*, perché richiede “un solo” operando.

Strutture di Commutazione

Sebbene la struttura `if` sia abbastanza versatile e sufficiente per controllare il flusso del programma, la struttura di controllo `switch` può essere più appropriata quando occorre valutare risultati diversi da vero o falso. Per esempio, se vogliamo intraprendere un'azione distinta per ogni elemento scelto da una lista, sarà necessario scrivere una struttura `if` per ogni valutazione:

```
// Available languages: en (English), es (Spanish), pt (Portuguese)
let language = "pt";

// Variable to register whether the language was found in the list
let found = 0;

if ( language == "en" )
{
    found = 1;
    console.log("English");
}

if ( found == 0 && language == "es" )
{
    found = 1;
    console.log("Spanish");
}

if ( found == 0 && language == "pt" )
{
    found = 1;
    console.log("Portuguese");
}

if ( found == 0 )
{
    console.log(language, " is unknown to me");
}
```

In questo esempio, una variabile ausiliaria `found` è usata da tutte le strutture `if` per scoprire se si è verificata una corrispondenza. In un caso come questo, la struttura `switch` svolgerà lo stesso compito, ma in modo più succinto:

```
switch ( language )
{
  case "en":
    console.log("English");
    break;
  case "es":
    console.log("Spanish");
    break;
  case "pt":
    console.log("Portuguese");
    break;
  default:
    console.log(language, " not found");
}
```

Ogni `case` annidato è chiamato *clausola*. Quando una clausola corrisponde all'espressione valutata, esegue le istruzioni che seguono i due punti fino all'istruzione `break`. L'ultima clausola non ha bisogno di un'istruzione `break` ed è spesso usata per impostare l'azione predefinita quando non ci sono altre corrispondenze. Come visto nell'esempio, la variabile ausiliaria non è necessaria nella struttura `switch`.

WARNING

`switch` utilizza il confronto rigoroso per abbinare le espressioni alle sue clausole `case`.

Se più di una clausola fa scattare la stessa azione, si possono combinare due o più condizioni `case`:

```
switch ( language )
{
    case "en":
    case "en_US":
    case "en_GB":
        console.log("English");
        break;
    case "es":
        console.log("Spanish");
        break;
    case "pt":
    case "pt_BR":
        console.log("Portuguese");
        break;
    default:
        console.log(language, " not found");
}
```

Cicli

Negli esempi precedenti, le strutture `if` e `switch` erano ben adatte a compiti che devono essere eseguiti solo una volta dopo aver passato uno o più test condizionali. Tuttavia, ci sono situazioni in cui un compito deve essere eseguito ripetutamente – in un cosiddetto *loop* – finché la sua espressione condizionale continua a risultare vera. Se avete bisogno di sapere se un numero è primo, per esempio, dovrete controllare se dividendo questo numero per qualsiasi intero maggiore di 1 e minore di se stesso si ha un resto uguale a 0. Se è così, il numero ha un fattore intero e non è primo. (Questo non è un metodo rigoroso o efficiente per trovare i numeri primi, ma funziona come semplice esempio). Le strutture di controllo dei loop sono più adatte a questi casi, in particolare l'istruzione `while`:

```
// A naive prime number tester

// The number we want to evaluate
let candidate = 231;

// Auxiliary variable
let is_prime = true;

// The first factor to try
let factor = 2;

// Execute the block statement if factor is
// less than candidate and keep doing it
// while factor is less than candidate
while ( factor < candidate )
{
    if ( candidate % factor == 0 )
    {
        // The remainder is zero, so the candidate is not prime
        is_prime = false;
        break;
    }

    // The next factor to try. Simply
    // increment the current factor by one
    factor++;
}

// Display the result in the console window.
// If candidate has no integer factor, then
// the auxiliary variable is_prime still true
if ( is_prime )
{
    console.log(candidate, "is prime");
}
else
{
    console.log(candidate, "is not prime");
}
```

La dichiarazione di blocco che segue l'istruzione `while` verrà eseguita ripetutamente finché la condizione `factor < candidate` è vera. Verrà eseguita almeno una volta, finché inizializzeremo la

variabile `factor` con un valore inferiore a `candidate`. La struttura `if` annidata nella struttura `while` valuterà se il resto di `candidate` diviso per `factor` è zero. Se è così, il numero `candidate` non è primo e il ciclo può finire. L'istruzione `break` terminerà il ciclo e l'esecuzione salterà alla prima istruzione dopo il blocco `while`.

Nota che il risultato della condizione usata dall'istruzione `while` deve cambiare a ogni ciclo, altrimenti l'istruzione di blocco andrà in loop “per sempre”. Nell'esempio, incrementiamo la variabile `factor`—il prossimo divisore che vogliamo provare—e questo garantisce che il ciclo finisca a un certo punto.

Questa semplice implementazione di un tester di numeri primi funziona come previsto. Tuttavia, sappiamo che un numero che non è divisibile per due non sarà divisibile per nessun altro numero pari. Pertanto, potremmo semplicemente saltare i numeri pari aggiungendo un'altra istruzione `if`:

```
while ( factor < candidate )
{
    // Skip even factors bigger than two
    if ( factor > 2 && factor % 2 == 0 )
    {
        factor++;
        continue;
    }

    if ( candidate % factor == 0 )
    {
        // The remainder is zero, so the candidate is not prime
        is_prime = false;
        break;
    }

    // The next number that will divide the candidate
    factor++;
}
```

L'istruzione `continue` è simile all'istruzione `break`, ma invece di finire questa iterazione del ciclo, ignorerà il resto del blocco del ciclo e inizierà una nuova iterazione. Nota che la variabile `factor` è stata modificata prima dell'istruzione `continue`, altrimenti il ciclo avrebbe lo stesso risultato nella prossima iterazione. Questo esempio è troppo semplice e saltare una parte del ciclo non migliorerà realmente le sue prestazioni, ma saltare le istruzioni ridondanti è molto importante quando si scrivono applicazioni efficienti.

I cicli sono così comunemente usati che esistono in molte varianti diverse. Il ciclo `for` è particolarmente adatto per iterare attraverso valori sequenziali, perché ci permette di definire le regole del ciclo in una sola riga:

```
for ( let factor = 2; factor < candidate; factor++ )
{
    // Skip even factors bigger than two
    if ( factor > 2 && factor % 2 == 0 )
    {
        continue;
    }

    if ( candidate % factor == 0 )
    {
        // The remainder is zero, so the candidate is not prime
        is_prime = false;
        break;
    }
}
```

Questo esempio produce esattamente lo stesso risultato dell'esempio precedente `while`, ma la sua espressione parentetica include tre parti, separate da punto e virgola: l'inizializzazione (`let factor = 2`), la condizione del ciclo (`factor < candidate`), e l'espressione finale da valutare alla fine di ogni iterazione del ciclo (`factor++`). Le istruzioni `continue` e `break` si applicano anche ai cicli `for`. L'espressione finale tra le parentesi (`factor++`) sarà valutata dopo l'istruzione `continue`, quindi non dovrebbe essere all'interno dell'istruzione di blocco, altrimenti sarà incrementata due volte prima della prossima iterazione.

JavaScript ha tipi speciali di cicli `for` per lavorare con oggetti simili ad array. Potremmo, per esempio, controllare un array di variabili invece di una sola:

```
// A naive prime number tester

// The array of numbers we want to evaluate
let candidates = [111, 139, 293, 327];

// Evaluates every candidate in the array
for (candidate of candidates)
{
    // Auxiliary variable
    let is_prime = true;

    for (let factor = 2; factor < candidate; factor++)
    {
        // Skip even factors bigger than two
        if (factor > 2 && factor % 2 == 0)
        {
            continue;
        }

        if (candidate % factor == 0)
        {
            // The remainder is zero, so the candidate is not prime
            is_prime = false;
            break;
        }
    }

    // Display the result in the console window
    if (is_prime)
    {
        console.log(candidate, "is prime");
    }
    else
    {
        console.log(candidate, "is not prime");
    }
}
```

L'istruzione `for (candidate of candidates)` assegna un elemento dell'array `candidates` alla variabile `candidate` e lo usa nella dichiarazione di blocco, ripetendo il processo per ogni elemento dell'array. Non c'è bisogno di dichiarare separatamente `candidate`, perché il ciclo `for` lo definisce. Infine, lo stesso codice dell'esempio precedente è stato annidato in questa nuova istruzione a blocchi, ma questa volta testando ogni `candidate` nell'array.

Esercizi Guidati

1. Quali valori della variabile `my_var` corrispondono alla condizione `my_var > 0 && my_var < 9?`

2. Quali valori della variabile `my_var` corrispondono alla condizione `my_var > 0 && my_var < 9?`

3. Quante volte il seguente ciclo `while` esegue la sua dichiarazione di blocco?

```
let i = 0;
while ( 1 )
{
    if ( i == 10 )
    {
        continue;
    }
    i++;
}
```

Esercizi Esplorativi

1. Cosa succede se l'operatore di assegnazione uguale `=` viene usato al posto dell'operatore di confronto uguale `==`?

2. Scrivi un frammento di codice usando la struttura di controllo `if` dove un confronto ordinario di uguaglianza restituirà `true`, ma un confronto di uguaglianza rigoroso `no`.

3. Riscrivi la seguente istruzione `for` usando l'operatore logico unario `NOT` nella condizione del ciclo. Il risultato della condizione dovrebbe essere lo stesso.

```
for ( let factor = 2; factor < candidate; factor++ )
```

4. Sulla base degli esempi di questa lezione, scrivi una struttura di controllo del loop che mostri tutti i fattori interi di un dato numero.

Sommario

Questa lezione spiega come usare le strutture di controllo nel codice JavaScript. Le strutture condizionali e i loop sono elementi essenziali di qualsiasi paradigma di programmazione, e lo sviluppo web in JavaScript non fa eccezione. La lezione passa in rassegna i seguenti concetti e procedure:

- L'istruzione `if` e gli operatori di confronto.
- Come usare la struttura `switch` con `case`, `default` e `break`.
- La differenza tra confronto ordinario e rigoroso.
- Strutture di controllo dei cicli: `while` e `for`.

Risposte agli Esercizi Guidati

- Quali valori della variabile `my_var` corrispondono alla condizione `my_var > 0 && my_var < 9?`

Solo i numeri che sono sia maggiori di 0 che minori di 9. L'operatore logico `&&` (AND) richiede che entrambi i confronti corrispondano.

- Quali valori della variabile `my_var` corrispondono alla condizione `my_var > 0 && my_var < 9?`

Solo i numeri che sono sia maggiori di 0 che minori di 9. L'operatore logico `&&` (AND) richiede che entrambi i confronti corrispondano.

- Quante volte il seguente ciclo `while` esegue la sua dichiarazione di blocco?

```
let i = 0;
while (1)
{
    if (i == 10)
    {
        continue;
    }
    i++;
}
```

La dichiarazione di blocco si ripeterà indefinitamente, poiché non è stata fornita alcuna condizione di arresto.

Risposte agli Esercizi Esplorativi

- Cosa succede se l'operatore di assegnazione uguale `=` viene usato al posto dell'operatore di confronto uguale `==`?

Il valore a destra dell'operatore viene assegnato alla variabile a sinistra e il risultato viene passato al confronto, che potrebbe non essere il comportamento desiderato.

- Scrivi un frammento di codice usando la struttura di controllo `if` dove un confronto ordinario di uguaglianza restituirà `true`, ma un confronto di uguaglianza rigoroso `no`.

```
let a = "1";
let b = 1;

if ( a == b )
{
    console.log("An ordinary comparison will match.");
}

if ( a === b )
{
    console.log("A strict comparison will not match.");
}
```

- Riscrivi la seguente istruzione `for` usando l'operatore logico unario `NOT` nella condizione del ciclo. Il risultato della condizione dovrebbe essere lo stesso.

```
for ( let factor = 2; factor < candidate; factor++ )
```

Risposta:

```
for ( let factor = 2; ! (factor >= candidate); factor++ )
```

- Sulla base degli esempi di questa lezione, scrivi una struttura di controllo del loop che mostri tutti i fattori interi di un dato numero.

```
for ( let factor = 2; factor <= my_number; factor++ )
{
  if ( my_number % factor == 0 )
  {
    console.log(factor, " is an integer factor of ", my_number);
  }
}
```



034.3 Lezione 2

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	034 Programmazione JavaScript
Obiettivo:	034.3 Strutture di Controllo e Funzioni in JavaScript
Lezione:	2 di 2

Introduzione

Oltre all'insieme standard di funzioni integrate fornite dal linguaggio JavaScript, gli sviluppatori possono scrivere le proprie funzioni personalizzate per mappare un input su un output adatto alle esigenze dell'applicazione. Le funzioni personalizzate sono fondamentalmente un insieme di istruzioni incapsulate per essere usate altrove come parte di un'espressione.

Usare le funzioni è un buon modo per evitare di scrivere codice duplicato, perché esse possono essere chiamate da diversi punti del programma. Inoltre, raggruppare le istruzioni in funzioni facilita il collegamento delle azioni personalizzate agli eventi, che è un aspetto centrale della programmazione JavaScript.

Definire una Funzione

Man mano che un programma cresce diventa più difficile organizzare ciò che fa senza usare funzioni. Ogni funzione ha il suo ambito “privato” di variabili, così le variabili definite all'interno di una funzione saranno disponibili solo all'interno di quella stessa funzione. Così facendo si ridurranno i rischi che si faccia confusione con le variabili di altre funzioni. Le variabili globali sono ancora

accessibili dall'interno delle funzioni, ma il modo preferibile per inviare valori di input a una funzione è attraverso i *parametri di funzione*. Come esempio, ci baseremo sul validatore di numeri primi della lezione precedente:

```
// A naive prime number tester

// The number we want to evaluate
let candidate = 231;

// Auxiliary variable
let is_prime = true;

// Start with the lowest prime number after 1
let factor = 2;

// Keeps evaluating while factor is less than the candidate
while ( factor < candidate )
{
    if ( candidate % factor == 0 )
    {
        // The remainder is zero, so the candidate is not prime
        is_prime = false;
        break;
    }

    // The next number that will divide the candidate
    factor++;
}

// Display the result in the console window
if ( is_prime )
{
    console.log(candidate, "is prime");
}
else
{
    console.log(candidate, "is not prime");
}
```

Se più avanti nel codice hai bisogno di controllare se un numero è primo, sarebbe necessario ripetere il codice che è già stato scritto. Questa pratica non è raccomandata, perché qualsiasi correzione o miglioramento al codice originale avrebbe bisogno di essere replicato manualmente ovunque il

codice sia stato copiato. Inoltre, ripetere il codice grava sul browser e sulla rete, possibilmente rallentando la visualizzazione della pagina web. Invece di fare questo, spostate le dichiarazioni appropriate in una funzione:

```
// A naive prime number tester function
function test_prime(candidate)
{
    // Auxiliary variable
    let is_prime = true;

    // Start with the lowest prime number after 1
    let factor = 2;

    // Keeps evaluating while factor is less than the candidate
    while ( factor < candidate )
    {

        if ( candidate % factor == 0 )
        {
            // The remainder is zero, so the candidate is not prime
            is_prime = false;
            break;
        }

        // The next number that will divide the candidate
        factor++;
    }

    // Send the answer back
    return is_prime;
}
```

La dichiarazione di funzione inizia con un'istruzione `function`, seguita dal nome della funzione e dai suoi parametri. Il nome della funzione deve seguire le stesse regole dei nomi delle variabili. I parametri della funzione, noti anche come *argomenti* della funzione, sono separati da virgole e racchiusi da parentesi.

TIP

Elencare gli argomenti nella dichiarazione della funzione non è obbligatorio. Gli argomenti passati a una funzione possono essere recuperati da un oggetto `arguments` simile a un array all'interno della funzione stessa. L'indice degli argomenti inizia da 0, quindi il primo argomento è `arguments[0]`, il secondo argomento è `arguments[1]`, e così via.

Nell'esempio la funzione `test_prime` ha un solo argomento: l'argomento `candidate`, che è il candidato numero primo da testare. Gli argomenti delle funzioni operano come variabili, ma i loro valori sono assegnati dall'istruzione che chiama la funzione. Per esempio, l'istruzione `test_prime(231)` richiamerà la funzione `test_prime` e assegnerà il valore 231 all'argomento `candidate`, che sarà poi disponibile nel corpo della funzione come una normale variabile.

Se l'istruzione chiamante usa variabili semplici per i parametri della funzione, i loro valori saranno copiati negli argomenti della funzione. Questa procedura—copiare i valori dei parametri usati nell'istruzione chiamante nei parametri usati all'interno della funzione—è chiamata *passare gli argomenti per valore*. Qualsiasi modifica apportata dalla funzione all'argomento non influisce sulla variabile originale usata nell'istruzione chiamante. Tuttavia, se l'istruzione chiamante usa oggetti complessi come argomenti (cioè un oggetto con proprietà e metodi collegati a esso) per i parametri della funzione, essi saranno *passati come riferimento* e la funzione potrà modificare l'oggetto originale usato nell'istruzione chiamante.

Gli argomenti che sono passati per valore, così come le variabili dichiarate all'interno della funzione, non sono visibili al di fuori di essa. Cioè, il loro ambito è limitato al corpo della funzione in cui sono stati dichiarati. Ciononostante, le funzioni sono solitamente impiegate per creare qualche output visibile al di fuori della funzione. Per condividere un valore con la sua funzione chiamante, una funzione definisce una dichiarazione `return`.

Per esempio, la funzione `test_prime` nell'esempio precedente restituisce il valore della variabile `is_prime`. Pertanto, la funzione può sostituire la variabile ovunque sarebbe usata nell'esempio originale:

```
// The number we want to evaluate
let candidate = 231;

// Display the result in the console window
if ( test_prime(candidate) )
{
    console.log(candidate, "is prime");
}
else
{
    console.log(candidate, "is not prime");
}
```

L'istruzione `return`, come indica il suo nome, restituisce il controllo alla funzione chiamante. Pertanto, ovunque l'istruzione `return` sia posizionata nella funzione, nulla di ciò che la segue viene eseguito. Una funzione può contenere più dichiarazioni `return`. Questa pratica può essere utile se

alcune sono all'interno di blocchi condizionali di istruzioni, in modo che la funzione possa o meno eseguire una particolare istruzione `return` a ogni esecuzione.

Alcune funzioni possono non restituire un valore, quindi la dichiarazione `return` non è obbligatoria. Le istruzioni interne della funzione vengono eseguite indipendentemente dalla sua presenza, quindi le funzioni, per esempio, possono anche essere usate per cambiare i valori delle variabili globali o il contenuto degli oggetti passati per riferimento. Nonostante ciò, se la funzione non ha una dichiarazione `return`, il suo valore di ritorno di default è impostato a `undefined`: una variabile riservata che non ha un valore e non può essere scritta.

Espressioni di Funzione

In JavaScript, le funzioni sono solo un altro tipo di *oggetto*. Così, le funzioni possono essere impiegate in uno script come variabili. Questa caratteristica diventa esplicita quando la funzione è dichiarata usando una sintassi alternativa, chiamata *espressione di funzione*:

```
let test_prime = function(candidate)
{
    // Auxiliary variable
    let is_prime = true;

    // Start with the lowest prime number after 1
    let factor = 2;

    // Keeps evaluating while factor is less than the candidate
    while ( factor < candidate )
    {

        if ( candidate % factor == 0 )
        {
            // The remainder is zero, so the candidate is not prime
            is_prime = false;
            break;
        }

        // The next number that will divide the candidate
        factor++;
    }

    // Send the answer back
    return is_prime;
}
```

L'unica differenza tra questo esempio e la dichiarazione di funzione nell'esempio precedente è nella prima riga: `let test_prime = function(candidate)` invece di `function test_prime(candidate)`. In un'espressione di funzione, il nome `test_prime` è usato per l'oggetto che contiene la funzione e non per nominare la funzione stessa. Le funzioni definite nelle espressioni di funzione sono chiamate allo stesso modo delle funzioni definite usando la sintassi di dichiarazione. Tuttavia, mentre le funzioni dichiarate possono essere chiamate prima o dopo la loro dichiarazione, le espressioni di funzione possono essere chiamate solo dopo la loro inizializzazione. Come per le variabili, chiamare una funzione definita in un'espressione prima della sua inizializzazione causerà un errore di riferimento.

Ricorsione di Funzione

Oltre a eseguire dichiarazioni e chiamare funzioni integrate, le funzioni personalizzate possono anche chiamare altre funzioni personalizzate, comprese se stesse. Chiamare una funzione da se stessa è chiamata *ricorsione di funzione*. A seconda del tipo di problema che state cercando di risolvere, l'uso di funzioni ricorsive può essere più semplice dell'uso di cicli annidati per eseguire compiti ripetitivi.

Finora sappiamo come usare una funzione per verificare se un dato numero è primo. Ora supponiamo che vogliate trovare il primo successivo a un dato numero. Potreste utilizzare un ciclo `while` per incrementare il numero candidato e scrivere un ciclo annidato che cercherà i fattori interi per quel candidato:

```
// This function returns the next prime number
// after the number given as its only argument
function next_prime(from)
{
    // We are only interested in the positive primes,
    // so we will consider the number 2 as the next
    // prime after any number less than two.
    if ( from < 2 )
    {
        return 2;
    }

    // The number 2 is the only even positive prime,
    // so it will be easier to treat it separately.
    if ( from == 2 )
    {
        return 3;
    }

    // Decrement "from" if it is an even number
    if ( from % 2 == 0 )
```

```

{
  from--;
}

// Start searching for primes greater than 3.

// The prime candidate is the next odd number
let candidate = from + 2;

// "true" keeps the loop going until a prime is found
while ( true )
{
  // Auxiliary control variable
  let is_prime = true;

  // "candidate" is an odd number, so the loop will
  // try only the odd factors, starting with 3
  for ( let factor = 3; factor < candidate; factor = factor + 2 )
  {
    if ( candidate % factor == 0 )
    {
      // The remainder is zero, so the candidate is not prime.
      // Test the next candidate
      is_prime = false;
      break;
    }
  }
  // End loop and return candidate if it is prime
  if ( is_prime )
  {
    return candidate;
  }
  // If prime not found yet, try the next odd number
  candidate = candidate + 2;
}

let from = 1024;
console.log("The next prime after", from, "is", next_prime(from));

```

Nota che abbiamo bisogno di usare una condizione costante per il ciclo while (l'espressione true dentro la parentesi) e la variabile ausiliaria `is_prime` per sapere quando fermare il ciclo. Anche se questa soluzione è corretta, l'uso di loop annidati non è così elegante come l'uso della ricorsione per eseguire lo stesso compito:

```
// This function returns the next prime number
// after the number given as its only argument
function next_prime(from)
{
    // We are only interested in the positive primes,
    // so we will consider the number 2 as the next
    // prime after any number less than two.
    if ( from < 2 )
    {
        return 2;
    }

    // The number 2 is the only even positive prime,
    // so it will be easier to treat it separately.
    if ( from == 2 )
    {
        return 3;
    }

    // Decrement "from" if it is an even number
    if ( from % 2 == 0 )
    {
        from--;
    }

    // Start searching for primes greater then 3.

    // The prime candidate is the next odd number
    let candidate = from + 2;

    // "candidate" is an odd number, so the loop will
    // try only the odd factors, starting with 3
    for ( let factor = 3; factor < candidate; factor = factor + 2 )
    {
        if ( candidate % factor == 0 )
        {
            // The remainder is zero, so the candidate is not prime.
            // Call the next_prime function recursively, this time
            // using the failed candidate as the argument.
            return next_prime(candidate);
        }
    }

    // "candidate" is not divisible by any integer factor other
```

```
// than 1 and itself, therefore it is a prime number.  
    return candidate;  
}  
  
let from = 1024;  
console.log("The next prime after", from, "is", next_prime(from));
```

Entrambe le versioni di `next_prime` restituiscono il prossimo numero primo dopo il numero dato come unico argomento (`from`). La versione ricorsiva, come quella precedente, inizia controllando i casi speciali (cioè i numeri minori o uguali a due). Poi incrementa il `candidate` e inizia a cercare qualsiasi fattore intero con il ciclo `for` (si noti che il ciclo `while` non è più presente). A quel punto, l'unico numero primo pari è già stato testato, quindi il candidato e i suoi possibili fattori vengono incrementati di due (un numero dispari più due è il prossimo numero dispari).

Ci sono solo due modi per uscire dal ciclo `for` dell'esempio. Se tutti i possibili fattori vengono testati e nessuno di essi ha un resto uguale a zero quando si divide il `candidate`, il ciclo `for` si completa e la funzione restituisce il candidato come il prossimo numero primo dopo `from`. Altrimenti, se `factor` è un fattore intero di `candidate` (`candidate % factor == 0`), il valore restituito viene dalla funzione `next_prime` chiamata ricorsivamente, questa volta con il `candidate` incrementato come parametro `from`. Le chiamate per `next_prime` saranno impilate l'una sull'altra, fino a quando un `candidate` non troverà nessun fattore intero. Allora l'ultima istanza di `next_prime` che contiene il numero primo lo restituirà alla precedente istanza di `next_prime`, e quindi successivamente fino alla prima istanza di `next_prime`. Anche se ogni invocazione della funzione usa gli stessi nomi per le variabili, le invocazioni sono isolate l'una dall'altra, quindi le loro variabili sono tenute separate in memoria.

Esercizi Guidati

1. Che tipo di sovraccarico può essere mitigato dagli sviluppatori usando le funzioni?

2. Qual è la differenza tra gli argomenti di funzione passati per valore e gli argomenti di funzione passati per riferimento?

3. Quale valore sarà usato come output di una funzione personalizzata se questa non ha una dichiarazione di ritorno?

Esercizi Esplorativi

1. Qual è la probabile causa di un *Uncaught Reference Error* registrato quando si chiama una funzione dichiarata con la sintassi *expression*?

2. Scrivi una funzione chiamata `multiples_of` che riceve tre argomenti: `factor`, `from` e `to`. All'interno della funzione, usa l'istruzione `console.log()` per visualizzare tutti i multipli di `factor` che si trovano tra `from` e `to`.

Sommario

Questa lezione spiega come scrivere funzioni personalizzate nel codice JavaScript. Le funzioni personalizzate permettono allo sviluppatore di dividere l'applicazione in "pezzi" di codice riutilizzabile, rendendo più facile scrivere e mantenere programmi più grandi. La lezione passa in rassegna i seguenti concetti e procedure:

- Come definire una funzione personalizzata: dichiarazioni di funzione ed espressioni di funzione.
- Usare i parametri come input della funzione.
- Usare l'istruzione `return` per impostare l'output della funzione.
- Ricorsione delle funzioni.

Risposte agli Esercizi Guidati

1. Che tipo di sovraccarico può essere mitigato dagli sviluppatori usando le funzioni?

Le funzioni ci permettono di riutilizzare il codice, il che facilita la sua manutenzione. Un file di script più piccolo fa anche risparmiare memoria e tempo di download.

2. Qual è la differenza tra gli argomenti di funzione passati per valore e gli argomenti di funzione passati per riferimento?

Quando viene passato per valore, l'argomento viene copiato nella funzione e la funzione non è in grado di modificare la variabile originale nell'istruzione chiamante. Quando viene passato per riferimento, la funzione è in grado di manipolare la variabile originale usata nell'istruzione chiamante.

3. Quale valore sarà usato come output di una funzione personalizzata se questa non ha una dichiarazione di ritorno?

Il valore restituito sarà impostato su `undefined`.

Risposte agli Esercizi Esplorativi

- Qual è la probabile causa di un *Uncaught Reference Error* registrato quando si chiama una funzione dichiarata con la sintassi *expression*?

La funzione è stata chiamata prima della sua dichiarazione nel file di script.

- Scrivi una funzione chiamata `multiples_of` che riceve tre argomenti: `factor`, `from` e `to`. All'interno della funzione, usa l'istruzione `console.log()` per visualizzare tutti i multipli di `factor` che si trovano tra `from` e `to`.

```
function multiples_of(factor, from, to)
{
    for ( let number = from; number <= to; number++ )
    {
        if ( number % factor == 0 )
        {
            console.log(factor, "x", number / factor, "=", number);
        }
    }
}
```



034.4 Manipolazione con JavaScript del Contenuto e dello Stile di un Sito Web

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 034.4

Peso

4

Arese di Conoscenza Chiave

- Comprendere il concetto e la struttura DOM
- Cambiare il contenuto e le proprietà degli elementi HTML attraverso DOM
- Cambiare lo stile CSS degli elementi HTML attraverso DOM
- Attivare funzioni JavaScript dagli elementi HTML

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- `document.getElementById()`, `document.getElementsByClassName()`,
`document.getElementsByTagName()`, `document.querySelectorAll()`,
`document.querySelector()`,
- `innerHTML`, `setAttribute()`, `removeAttribute()` proprietà e metodi degli elementi DOM
- Proprietà e metodi degli elementi DOM `classList`, `classList.add()`, `classList.remove()`,
`classList.toggle()`
- Attributi di elementi HTML `onClick`, `onMouseOver`, `onMouseOut`



034.4 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	034 Programmazione JavaScript
Obiettivo:	034.4 Manipolazione con JavaScript del Contenuto e dello Stile di un Sito Web
Lezione:	1 di 1

Introduzione

HTML, CSS e JavaScript sono tre tecnologie distinte che si incontrano sul Web. Per fare pagine veramente dinamiche e interattive, il programmatore JavaScript deve combinare componenti HTML e CSS in esecuzione. Un compito che è notevolmente facilitato dall'uso del *Document Object Model* (DOM).

Interagire con il DOM

Il DOM è una struttura di dati che funziona come un'interfaccia di programmazione al documento, dove ogni aspetto di quest'ultimo è rappresentato come un nodo nel DOM e ogni cambiamento fatto al DOM si riverbera immediatamente nel documento. Per mostrare come usare il DOM in JavaScript, salva il seguente codice HTML in un file chiamato `example.html`:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>HTML Manipulation with JavaScript</title>
</head>
<body>

<div class="content" id="content_first">
<p>The dynamic content goes here</p>
</div><!-- #content_first -->

<div class="content" id="content_second" hidden>
<p>Second section</p>
</div><!-- #content_second -->

</body>
</html>
```

Il DOM è disponibile solo dopo che l'HTML è stato caricato, quindi scrivi il seguente JavaScript alla fine del corpo della pagina (prima del tag finale `</body>`):

```
<script>
let body = document.getElementsByTagName("body")[0];
console.log(body.innerHTML);
</script>
```

L'oggetto `document` è l'elemento superiore del DOM, tutti gli altri elementi si diramano da esso. Il metodo `getElementsByName()` elenca tutti gli elementi discendenti da `document` che hanno il nome del tag dato. Anche se il tag `body` è usato solo una volta nel documento, il metodo `getElementsByName()` restituisce sempre una collezione simile a un array di elementi trovati, da cui l'uso dell'indice `[0]` per restituire il primo (e unico) elemento trovato.

Contenuto HTML

Come mostrato nell'esempio precedente, l'elemento DOM restituito dal `document.getElementsByTagName("body")[0]` è stato assegnato alla variabile `body`. La variabile `body` può quindi essere usata per manipolare l'elemento `body` della pagina, perché eredita tutti i metodi e gli attributi DOM da quell'elemento. Per esempio, la proprietà `innerHTML` contiene l'intero codice di markup HTML scritto all'interno dell'elemento corrispondente, quindi può essere usata per leggere il markup interno. La nostra chiamata `console.log(body.innerHTML)` stampa il contenuto all'interno

di `<body></body>` nella console web. La variabile può anche essere usata per sostituire quel contenuto, come in `body.innerHTML = "<p>Content erased</p>"`.

Piuttosto che modificare intere porzioni di marcatura HTML, è più pratico mantenere inalterata la struttura del documento e interagire semplicemente con i suoi elementi. Una volta che il documento è stato renderizzato dal browser, tutti gli elementi sono accessibili con i metodi DOM. È possibile, ad esempio, elencare e accedere a tutti gli elementi HTML utilizzando la stringa speciale `*` nel metodo `getElementsByTagName()` dell'oggetto `document`:

```
let elements = document.getElementsByTagName("*");
for ( element of elements )
{
  if ( element.id == "content_first" )
  {
    element.innerHTML = "<p>New content</p>";
  }
}
```

Questo codice metterà tutti gli elementi trovati in `document` nella variabile `elements`. La variabile `elements` è un oggetto simile a un array, quindi possiamo iterare attraverso ciascuno dei suoi elementi con un ciclo `for`. Se la pagina HTML dove viene eseguito questo codice ha un elemento con un attributo `id` impostato su `content_first` (vedi la pagina HTML di esempio mostrata all'inizio della lezione), l'istruzione `if` corrisponde a quell'elemento e il suo contenuto di markup sarà cambiato in `<p>New content</p>`. Nota che gli attributi di un elemento HTML nel DOM sono accessibili usando la *notazione puntata* delle proprietà degli oggetti JavaScript: quindi, `element.id` si riferisce all'attributo `id` dell'elemento corrente del ciclo `for`. Si potrebbe anche usare il metodo `getAttribute()`, come in `element.getAttribute("id")`.

Non è necessario iterare attraverso tutti gli elementi se si vuole ispezionare solo un sottoinsieme di essi. Per esempio, il metodo `document.getElementsByClassName()` limita gli elementi trovati a quelli che hanno una classe specifica:

```
let elements = document.getElementsByClassName("content");
for ( element of elements )
{
  if ( element.id == "content_first" )
  {
    element.innerHTML = "<p>New content</p>";
  }
}
```

Tuttavia, iterare attraverso molti elementi del documento usando un ciclo *non* è la migliore strategia quando si deve cambiare un elemento specifico nella pagina.

Selezionare Elementi Specifici

JavaScript fornisce metodi ottimizzati per selezionare l'esatto elemento su cui si vuole lavorare. Il ciclo precedente potrebbe essere interamente sostituito dal metodo `document.getElementById()`:

```
let element = document.getElementById("content_first");
element.innerHTML = "<p>New content</p>";
```

Ogni attributo `id` nel documento deve essere unico, quindi il metodo `document.getElementById()` restituisce solo un singolo oggetto DOM. Anche la dichiarazione della variabile `element` può essere omessa, perché JavaScript ci permette di concatenare direttamente i metodi:

```
document.getElementById("content_first").innerHTML = "<p>New content</p>";
```

Il metodo `getElementById()` è il metodo preferibile per localizzare elementi nel DOM, perché le sue prestazioni sono molto migliori dei metodi iterativi quando si lavora con documenti complessi. Tuttavia, non tutti gli elementi hanno un ID esplicito, e il metodo restituisce un valore `null` se nessun elemento corrisponde all'ID fornito (questo impedisce anche l'uso di attributi o funzioni concatenate, come la `innerHTML` usata nell'esempio precedente). Inoltre, è più pratico assegnare gli attributi ID solo ai componenti principali della pagina e poi usare i selettori CSS per localizzare i loro elementi figli.

I selettori, introdotti in una precedente lezione sui CSS, sono schemi che corrispondono a elementi nel DOM. Il metodo `querySelector()` restituisce il primo elemento corrispondente nell'albero del DOM, mentre `querySelectorAll()` restituisce tutti gli elementi che corrispondono al selettore specificato.

Nell'esempio precedente, il metodo `getElementById()` recupera l'elemento con l'ID `content_first`. Il metodo `querySelector()` può eseguire lo stesso compito:

```
document.querySelector("#content_first").innerHTML = "<p>New content</p>";
```

Poiché il metodo `querySelector()` usa la sintassi del selettore, l'ID fornito deve iniziare con un carattere hash. Se non viene trovato nessun elemento corrispondente, il metodo `querySelector()` restituisce `null`.

Nell'esempio precedente, l'intero contenuto del div `content_first` è sostituito dalla stringa di testo fornita. La stringa ha del codice HTML al suo interno, il che non è considerato una buona pratica. Devi stare attento quando aggiungi codice HTML *hard-coded* al codice JavaScript, perché il tracciamento degli elementi può diventare difficile quando sono richiesti cambiamenti alla struttura generale del documento.

I selettori non sono limitati all'ID dell'elemento. L'elemento interno `p` può essere indirizzato direttamente:

```
document.querySelector("#content_first p").innerHTML = "New content";
```

Il selettore `#content_first p` corrisponderà solo al primo elemento `p` all'interno del div `#content_first`. Funziona bene se vogliamo manipolare il primo elemento. Tuttavia, potremmo voler cambiare il secondo paragrafo:

```
<div class="content" id="content_first">
<p>Don't change this paragraph.</p>
<p>The dynamic content goes here.</p>
</div><!-- #content_first -->
```

In questo caso, possiamo usare la pseudo-classe `:nth-child(2)` per abbinare il secondo elemento `p`:

```
document.querySelector("#content_first p:nth-child(2)").innerHTML = "New content";
```

Il numero 2 in `p:nth-child(2)` indica il secondo paragrafo che corrisponde al selettore. Vedi la lezione sui selettori CSS per saperne di più sui selettori e su come usarli.

Lavorare con gli Attributi

La capacità di JavaScript di interagire con il DOM non è limitata alla manipolazione del contenuto. Infatti, l'uso più pervasivo di JavaScript nel browser è quello di modificare gli attributi degli elementi HTML esistenti.

Diciamo che la nostra pagina originale HTML di esempio ha ora tre sezioni di contenuto:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>HTML Manipulation with JavaScript</title>
</head>
<body>

<div class="content" id="content_first" hidden>
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second" hidden>
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third" hidden>
<p>Third section.</p>
</div><!-- #content_third -->

</body>
</html>
```

Potresti voler rendere visibile solo uno di essi alla volta, da qui l'attributo `hidden` in tutti i tag `div`. Questo è utile, per esempio, per mostrare solo un'immagine da una galleria di immagini. Per renderne visibile una al caricamento della pagina, aggiungi il seguente codice JavaScript alla pagina:

```
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
    case 0:
        content_visible = "#content_first";
        break;
    case 1:
        content_visible = "#content_second";
        break;
    case 2:
        content_visible = "#content_third";
        break;
}

document.querySelector(content_visible).removeAttribute("hidden");
```

L'espressione valutata dall'istruzione `switch` restituisce casualmente il numero 0, 1 o 2. Il selettore ID corrispondente viene quindi assegnato alla variabile `content_visible`, che è usata dal metodo `querySelector(content_visible)`. La chiamata concatenata `removeAttribute("hidden")` rimuove l'attributo `hidden` dall'elemento.

È possibile anche l'approccio opposto: Tutte le sezioni potrebbero essere inizialmente visibili (senza l'attributo `hidden`) e il programma JavaScript può poi assegnare l'attributo `hidden` a ogni sezione tranne quella in `content_visible`. Per fare ciò, è necessario iterare attraverso tutti gli elementi `div` del contenuto che sono diversi da quello scelto, il che può essere fatto usando il metodo `querySelectorAll()`:

```
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
    case 0:
        content_visible = "#content_first";
        break;
    case 1:
        content_visible = "#content_second";
        break;
    case 2:
        content_visible = "#content_third";
        break;
}

// Hide all content divs, except content_visible
for ( element of document.querySelectorAll(".content:not(#"+content_visible+")) )
{
    // Hidden is a boolean attribute, so any value will enable it
    element.setAttribute("hidden", "");
}
```

Se la variabile `content_visible` è stata impostata a `#content_first`, il selettore sarà `.content:not(#content_first)`, che si legge come tutti gli elementi che hanno la classe `content` tranne quelli che hanno l'ID `content_first`. Il metodo `setAttribute()` aggiunge o cambia gli attributi degli elementi HTML. Il suo primo parametro è il nome dell'attributo e il secondo è il valore dell'attributo.

Tuttavia, il modo corretto per cambiare l'aspetto degli elementi è con i CSS. In questo caso, possiamo impostare la proprietà CSS `display` su `hidden` e poi cambiarla in `block` usando JavaScript:

```
<style>
div.content { display: none }
</style>

<div class="content" id="content_first">
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second">
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third">
<p>Third section.</p>
</div><!-- #content_third -->

<script>
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
    break;
  case 1:
    content_visible = "#content_second";
    break;
  case 2:
    content_visible = "#content_third";
    break;
}
document.querySelector(content_visible).style.display = "block";
</script>
```

Le stesse buone pratiche che si applicano a “mescolare” tag HTML con JavaScript si applicano anche ai CSS: per questo motivo scrivere le proprietà CSS direttamente nel codice JavaScript non è raccomandato. Invece, le regole CSS dovranno essere scritte separatamente dal codice JavaScript. Il modo corretto per alternare lo stile visivo è quello di selezionare una classe CSS predefinita per l’elemento.

Lavorare con le Classi

Gli elementi possono avere più di una classe associata, rendendo più facile scrivere stili che possono essere aggiunti o rimossi quando necessario. Sarebbe estenuante cambiare molti attributi CSS direttamente in JavaScript, così si può creare una nuova classe CSS con quegli attributi e poi aggiungere la classe all'elemento. Gli elementi DOM hanno la proprietà `classList`, che può essere usata per visualizzare e manipolare le classi assegnate all'elemento corrispondente.

Per esempio, invece di cambiare la visibilità dell'elemento, possiamo creare una classe CSS aggiuntiva per evidenziare il nostro div `content`:

```
div.content {  
    border: 1px solid black;  
    opacity: 0.25;  
}  
  
div.content.highlight {  
    border: 1px solid red;  
    opacity: 1;  
}
```

Questo foglio di stile aggiungerà un sottile bordo nero e una semi-trasparenza a tutti gli elementi che hanno la classe `content`. Solo gli elementi che hanno anche la classe `highlight` saranno completamente opachi e avranno un sottile bordo rosso. Quindi, invece di cambiare direttamente le proprietà CSS come abbiamo fatto prima, possiamo usare il metodo `classList.add("highlight")` nell'elemento selezionato:

```
// Which content to highlight
let content_highlight;

switch ( Math.floor(Math.random() * 3) )
{
    case 0:
        content_highlight = "#content_first";
        break;
    case 1:
        content_highlight = "#content_second";
        break;
    case 2:
        content_highlight = "#content_third";
        break;
}

// Highlight the selected div
document.querySelector(content_highlight).classList.add("highlight");
```

Tutte le tecniche e gli esempi che abbiamo visto finora sono stati eseguiti alla fine del processo di caricamento della pagina, ma *non sono* limitati a questa fase. Ciò che rende JavaScript così utile agli sviluppatori Web è la sua capacità di reagire agli eventi sulla pagina, cosa che vedremo in seguito.

Gestori di eventi

Tutti gli elementi visibili della pagina sono suscettibili di eventi interattivi, come il clic o il movimento del mouse stesso. Possiamo associare azioni personalizzate a questi eventi, il che espande notevolmente ciò che un documento HTML può fare.

Probabilmente l'elemento HTML più ovvio che beneficia di un'azione associata è l'elemento `button`. Per mostrare come funziona, aggiungi tre pulsanti sopra il primo elemento `div` della pagina di esempio:

```
<p>
<button>First</button>
<button>Second</button>
<button>Third</button>
</p>

<div class="content" id="content_first">
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second">
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third">
<p>Third section.</p>
</div><!-- #content_third -->
```

I pulsanti non fanno nulla da soli, ma supponiamo di voler evidenziare il `div` corrispondente al pulsante premuto. Possiamo usare l'attributo `onClick` per associare un'azione a ogni pulsante:

```
<p>
<button
onClick="document.getElementById('content_first').classList.toggle('highlight')">Fi
rst</button>
<button
onClick="document.getElementById('content_second').classList.toggle('highlight')">S
econd</button>
<button
onClick="document.getElementById('content_third').classList.toggle('highlight')">Th
ird</button>
</p>
```

Il metodo `classList.toggle()` aggiunge la classe specificata all'elemento se non è presente, e la rimuove se già lo è. Se esegui l'esempio, noterai che più di un `div` può essere evidenziato allo stesso tempo. Per evidenziare *solo* il `div` corrispondente al pulsante premuto, è necessario rimuovere la classe `highlight` dagli altri elementi `div`. Tuttavia, se l'azione personalizzata è troppo lunga o comporta più di una linea di codice, è più pratico scrivere una funzione separata dall'elemento tag:

```
function highlight(id)
{
    // Remove the "highlight" class from all content elements
    for ( element of document.querySelectorAll(".content") )
    {
        element.classList.remove('highlight');
    }

    // Add the "highlight" class to the corresponding element
    document.getElementById(id).classList.add('highlight');
}
```

Come gli esempi precedenti, questa funzione può essere posta all'interno di un tag `<script>` o in un file JavaScript esterno associato al documento. La funzione `highlight` rimuove prima la classe `highlight` da tutti gli elementi `div` associati alla classe `content`, poi aggiunge la classe `highlight` all'elemento scelto. Ogni pulsante dovrebbe poi chiamare questa funzione dal suo attributo `onClick`, usando l'ID corrispondente come argomento della funzione:

```
<p>
<button onClick="highlight('content_first')">First</button>
<button onClick="highlight('content_second')">Second</button>
<button onClick="highlight('content_third')">Third</button>
</p>
```

Oltre all'attributo `onClick`, potremmo usare l'attributo `onMouseOver` (attivato quando il dispositivo di puntamento viene usato per spostare il cursore sull'elemento), l'attributo `onMouseOut` (attivato quando il dispositivo di puntamento non è più contenuto nell'elemento), ecc. Inoltre, i gestori di eventi non sono limitati ai pulsanti, quindi puoi assegnare azioni personalizzate a questi gestori di eventi per tutti gli elementi HTML visibili.

Esercizi Guidati

1. Usando il metodo `document.getElementById()`, come potresti inserire la frase “Dynamic content” nel contenuto interno dell’elemento il cui ID è `message`?

2. Qual è la differenza tra referenziare un elemento attraverso il suo ID usando il metodo `document.querySelector()` e farlo tramite il metodo `document.getElementById()`?

3. Qual è lo scopo del metodo `classList.remove()`?

4. Qual è il risultato dell’uso del metodo `myElement.classList.toggle("active")` se `myElement` non ha la classe `active` assegnata a esso?

Esercizi Esplorativi

1. Quale argomento al metodo `document.querySelectorAll()` gli farà imitare il metodo `document.getElementsByTagName("input")?`

2. Come si può usare la proprietà `classList` per elencare tutte le classi associate a un dato elemento?

Sommario

Questa lezione spiega come usare JavaScript per cambiare i contenuti HTML e le loro proprietà CSS usando il DOM (Document Object Model). Questi cambiamenti possono essere innescati da eventi utente, utile per creare interfacce dinamiche. La lezione passa attraverso i seguenti concetti e procedure:

- Come ispezionare la struttura del documento usando metodi come `document.getElementById()`, `document.getElementsByClassName()`, `document.getElementsByTagName()`, `document.querySelector()` e `document.querySelectorAll()`.
- Come cambiare il contenuto del documento con la proprietà `innerHTML`.
- Come aggiungere e modificare gli attributi degli elementi della pagina con i metodi `setAttribute()` e `removeAttribute()`.
- Il modo corretto di manipolare le classi degli elementi usando la proprietà `classList` e la sua relazione con gli stili CSS.
- Come legare funzioni a eventi del mouse in elementi specifici.

Risposte agli Esercizi Guidati

1. Usando il metodo `document.getElementById()`, come potresti inserire la frase “Dynamic content” nel contenuto interno dell’elemento il cui ID è `message`?

Può essere inserita con la proprietà `innerHTML`:

```
document.getElementById("message").innerHTML = "Dynamic content"
```

2. Qual è la differenza tra referenziare un elemento attraverso il suo ID usando il metodo `document.querySelector()` e farlo tramite il metodo `document.getElementById()`?

L’ID deve essere accompagnato dal carattere hash (#) nelle funzioni che usano i selettori, come `document.querySelector()`.

3. Qual è lo scopo del metodo `classList.remove()`?

Rimuove la classe (il cui nome è dato come argomento della funzione) dall’attributo `class` dell’elemento corrispondente.

4. Qual è il risultato dell’uso del metodo `myelement.classList.toggle("active")` se `myelement` non ha la classe `active` assegnata a esso?

Il metodo assegnerà la classe `active` a `myelement`.

Risposte agli Esercizi Esplorativi

1. Quale argomento al metodo `document.querySelectorAll()` gli farà imitare il metodo `document.getElementsByTagName("input")`?

L'uso di `document.querySelectorAll("input")` corrisponderà a tutti gli elementi `input` nella pagina, proprio come `document.getElementsByTagName("input")`.

2. Come si può usare la proprietà `classList` per elencare tutte le classi associate a un dato elemento?

La proprietà `classList` è un oggetto simile a un array, quindi un ciclo `for` può essere usato per iterare tutte le classi che contiene.



Argomento 035: Programmazione Server con Node.js



035.1 Fondamenti di Node.js

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 035.1

Peso

1

Arese di Conoscenza Chiave

- Capire i concetti di Node.js
- Eseguire un'applicazione NodeJS
- Installare i pacchetti NPM

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- node [file.js]
- npm init
- npm install [module_name]
- package.json
- node_modules



035.1 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	035 Programmazione Server con Node.js
Obiettivo:	035.1 Fondamenti di Node.js
Lezione:	1 di 1

Introduzione

Node.js è un ambiente *runtime* JavaScript che consente di eseguire codice JavaScript sui server web – il cosiddetto web *backend* (lato server) – invece di usare un secondo linguaggio come Python o Ruby per programmi lato server. Il linguaggio JavaScript è già usato nel moderno lato frontend delle applicazioni web, interagendo con l'HTML e il CSS dell'interfaccia con cui l'utente interagisce in un browser web. Usare Node.js in tandem con JavaScript nel browser offre la possibilità di un solo linguaggio di programmazione per l'intera applicazione.

Il motivo principale dell'esistenza di Node.js è il modo in cui gestisce più connessioni simultanee nel back-end. Uno dei modi più comuni in cui un server di applicazioni Web gestisce le connessioni è l'esecuzione di più processi. Quando si apre un'applicazione desktop nel computer, viene avviato un processo che utilizza molte risorse. Ora pensa a quando migliaia di utenti stanno facendo la stessa cosa in una grande applicazione web.

Node.js evita questo problema usando un design chiamato *event loop*, che è un ciclo interno che controlla continuamente i compiti in arrivo da calcolare. Grazie all'uso diffuso di JavaScript e all'ubiquità delle tecnologie web, Node.js ha visto un'enorme adozione in applicazioni piccole e grandi. Ci sono anche altre caratteristiche che hanno aiutato Node.js a essere ampiamente adottato,

come l'elaborazione asincrona e non bloccante di input/output (I/O), che è spiegata più avanti in questa lezione.

L'ambiente Node.js utilizza un motore JavaScript per interpretare ed eseguire il codice JavaScript sul lato server o sul lato client. In queste condizioni, il codice JavaScript che il programmatore scrive viene analizzato e compilato *just-in-time* per eseguire le istruzioni macchina generate dal codice JavaScript originale.

NOTE Man mano che procederai con queste lezioni su Node.js, potrai notare che il JavaScript di Node.js *non è* esattamente lo stesso di quello che viene eseguito sul browser (che segue le specifiche ECMAScript), ma è comunque abbastanza simile.

Per Iniziare

Questa sezione e gli esempi seguenti presuppongono che Node.js sia già installato sul sistema operativo Linux e che l'utente abbia già delle competenze di base come l'esecuzione di comandi nel terminale.

Per eseguire i seguenti esempi, crea una directory di lavoro chiamata `node_examples`. Apri un prompt del terminale e digita `node`. Se hai installato correttamente Node.js, ti presenterà un prompt `>` dove puoi testare i comandi JavaScript in modo interattivo. Questo tipo di ambiente è chiamato REPL, che sta per “read, evaluate, print, and loop”. Digita il seguente input (o qualche altra istruzione JavaScript) al prompt `>`. Premi il tasto *Invio* dopo ogni riga, e l'ambiente REPL restituirà i risultati delle sue azioni:

```
> let array = ['a', 'b', 'c', 'd'];
undefined
> array.map( element, index ) => (`Element: ${element} at index: ${index}`);
[
  'Element: a at index: 0',
  'Element: b at index: 1',
  'Element: c at index: 2',
  'Element: d at index: 3'
]
>
```

Il codice è stato scritto usando la sintassi ES6, che offre una funzione `map` per iterare sull'array e visualizzare i risultati usando modelli di stringa. Puoi scrivere praticamente qualsiasi comando che sia valido. Per uscire dal terminale Node.js, digita `.exit`, ricordandoti di includere il punto iniziale.

Per script e moduli più lunghi, è più conveniente usare un editor di testo come VS Code, Emacs o

Vim. Puoi salvare le due righe di codice appena mostrate (con una piccola modifica) in un file chiamato `start.js`:

```
let array = ['a', 'b', 'c', 'd'];
array.map( (element, index) => ( console.log(`Element: ${element} at index: ${index}`)));
```

Poi puoi eseguire lo script dalla shell per produrre gli stessi risultati di prima:

```
$ node ./start.js
Element: a at index: 0
Element: b at index: 1
Element: c at index: 2
Element: d at index: 3
```

Prima di immergerti un po' di più nel codice, facciamo una panoramica di come funziona Node.js, usando il suo ambiente di esecuzione a *thread* singolo e il ciclo degli eventi (*event loop*).

Ciclo degli Eventi e Thread Singolo

È difficile dire quanto tempo impiegherà un programma Node.js per gestire una richiesta. Alcune richieste possono essere brevi—forse solo un ciclo attraverso le variabili in memoria e la loro restituzione—mentre altre possono necessitare di attività che richiedono tempo, come l’apertura di un file sul sistema o l’esecuzione di una query su un database e l’attesa dei risultati. Come fa Node.js a gestire questa incertezza? L’*event loop* è la risposta.

Immagina uno chef che svolge più compiti. Cuocere una torta è un compito che richiede molto tempo al forno. Lo chef non rimane lì ad aspettare che la torta sia pronta per mettersi a preparare il caffè. Invece, mentre il forno cuoce la torta, il cuoco prepara il caffè e svolge altri compiti in parallelo. Ma il cuoco controlla sempre se è il momento giusto per spostare l’attenzione su un compito specifico (preparare il caffè), o per tirare fuori la torta dal forno.

Il ciclo degli eventi è come lo chef che è costantemente consapevole delle attività circostanti. In Node.js, un “event-checker” controlla sempre le operazioni che si sono completate o sono in attesa di essere processate dal motore JavaScript.

Usando questo approccio un’operazione asincrona e lunga non blocca altre operazioni veloci che vengono dopo. Questo perché il meccanismo del ciclo degli eventi controlla sempre se quel compito lungo, come un’operazione di I/O, è già stato eseguito. In caso contrario, Node.js può continuare ad elaborare altri compiti. Una volta che il compito in background è completato, i risultati vengono

restituiti e l'applicazione che usa Node.js può utilizzare una funzione di *trigger* (callback) per elaborare ulteriormente l'output.

Poiché Node.js evita l'uso di thread multipli, come fanno altri ambienti, è chiamato un *ambiente single-threaded*, e un approccio *non bloccante* è della massima importanza. Questo è il motivo per cui Node.js utilizza un ciclo di eventi. Per compiti ad alta intensità di calcolo, Node.js non è comunque tra gli strumenti migliori: ci sono altri linguaggi di programmazione e ambienti che affrontano questi problemi in modo più efficiente.

Nelle sezioni seguenti daremo uno sguardo più da vicino alle funzioni di callback. Per ora, ti basti sapere che tali funzioni sono trigger che vengono eseguiti al completamento di un'operazione predefinita.

Moduli

È buona pratica suddividere funzionalità complesse e parti di codice esteso in parti più piccole. Questa modularizzazione aiuta a organizzare meglio la base di codice, ad astrarre le implementazioni e a evitare complicati problemi di ingegneria software. Per soddisfare queste necessità, i programmatore confezionano blocchi di codice sorgente perché siano elaborati da altre parti interne o esterne di codice.

Consideriamo l'esempio di un programma che calcola il volume di una sfera. Apri il tuo editor di testo e crea un file chiamato `volumeCalculator.js` contenente il seguente JavaScript:

```
const sphereVol = (radius) => {
  return 4 / 3 * Math.PI * radius
}

console.log('A sphere with radius 3 has a ${sphereVol(3)} volume.');
console.log('A sphere with radius 6 has a ${sphereVol(6)} volume.');
```

Ora, esegui il file usando Node:

```
$ node volumeCalculator.js
A sphere with radius 3 has a 113.09733552923254 volume.
A sphere with radius 6 has a 904.7786842338603 volume.
```

È stata usata una semplice funzione per calcolare il volume di una sfera, in base al suo raggio. Immaginiamo di dover calcolare anche il volume di un cilindro, di un cono, e così via: notiamo subito che queste funzioni specifiche devono essere aggiunte al file `volumeCalculator.js`, che può

diventare un'enorme collezione di funzioni. Per organizzare meglio la struttura, possiamo suddividere il tutto in pacchetti di codice separato attraverso la creazione di moduli.

Per farlo, crea un file separato chiamato `polyhedrons.js` realizzato nel seguente modo:

```
const coneVol = (radius, height) => {
  return 1 / 3 * Math.PI * Math.pow(radius, 2) * height;
}

const cylinderVol = (radius, height) => {
  return Math.PI * Math.pow(radius, 2) * height;
}

const sphereVol = (radius) => {
  return 4 / 3 * Math.PI * Math.pow(radius, 3);
}

module.exports = {
  coneVol,
  cylinderVol,
  sphereVol
}
```

Ora, nel file `volumeCalculator.js`, elimina il vecchio codice e sostituiscilo con:

```
const polyhedrons = require('./polyhedrons.js');

console.log('A sphere with radius 3 has a ${polyhedrons.sphereVol(3)} volume.');
console.log('A cylinder with radius 3 and height 5 has a ${polyhedrons.cylinderVol(3, 5)} volume.');
console.log('A cone with radius 3 and height 5 has a ${polyhedrons.coneVol(3, 5)} volume.');
```

E poi esegui l'ambiente Node.js:

```
$ node volumeCalculator.js
A sphere with radius 3 has a 113.09733552923254 volume.
A cylinder with radius 3 and height 5 has a 141.3716694115407 volume.
A cone with radius 3 and height 5 has a 47.12388980384689 volume.
```

Nell'ambiente Node.js, ogni file di codice sorgente è considerato un modulo, ma il termine “modulo”

in Node.js indica un pacchetto di codice strutturato come nell'esempio precedente. Usando i moduli, abbiamo astratto le funzioni del volume dal file principale, `volumeCalculator.js`, riducendo così la sua dimensione e rendendo più facile applicare i test unitari, che sono una buona pratica quando si sviluppano applicazioni del mondo reale.

Ora che sappiamo come si usano i moduli in Node.js, possiamo usare uno degli strumenti più importanti: il *Node Package Manager* (NPM).

Uno dei compiti principali di NPM è quello di gestire, scaricare e installare moduli esterni nel progetto o nel sistema operativo. Puoi inizializzare un repository di node con il comando `npm init`.

NPM porrà le domande predefinite sul nome del tuo repository, la versione, la descrizione e così via. Puoi saltare questi passaggi usando `npm init --yes`, e il comando genererà automaticamente un file `package.json` che descrive le proprietà del tuo progetto/modulo.

Apri il file `package.json` nel tuo editor di testo preferito e vedrai un file JSON contenente proprietà come parole chiave, comandi di script da usare con NPM, un nome e così via.

Una di queste proprietà indica le dipendenze che sono installate nel tuo repository locale. NPM aggiungerà il nome e la versione di queste dipendenze in `package.json`, insieme a `package-lock.json`, un altro file usato come *fallback* da NPM nel caso in cui il file `package.json` fallisca.

Digita quanto segue sul tuo terminale:

```
$ npm i dayjs
```

L'argomento `i` è una abbreviazione che sta per `install`. Se sei connesso ad Internet, NPM cercherà un modulo chiamato `dayjs` nel repository remoto di Node.js, scaricherà il modulo e lo installerà localmente. NPM aggiungerà anche questa dipendenza ai tuoi file `package.json` e `package-lock.json`. Ora puoi vedere che c'è una cartella chiamata `node_modules`, che contiene il modulo installato insieme ad altri moduli se necessari. La cartella `node_modules` contiene il codice effettivo che verrà usato quando la libreria verrà importata e chiamata. Tuttavia, questa cartella non viene salvata nei sistemi di *versioning* che usano Git, poiché il file `package.json` fornisce tutte le dipendenze utilizzate. Un altro utente può prendere il file `package.json` ed eseguire semplicemente `npm install` nella propria macchina, dove NPM creerà una cartella `node_modules` con tutte le dipendenze nel `package.json`, evitando così il controllo di versione per le migliaia di file disponibili sul repository NPM.

Ora che il modulo `dayjs` è installato nella directory locale, apri la console di Node.js e digita le seguenti righe:

```
const dayjs = require('dayjs');
dayjs().format('YYYY MM-DDTHH:mm:ss')
```

Il modulo `dayjs` è caricato con la parola chiave `require`. Quando viene chiamato un metodo dal modulo, la libreria prende il datetime corrente del sistema e lo restituisce nel formato specificato:

```
2020 11-22T11:04:36
```

Questo è lo stesso meccanismo usato nell'esempio precedente, dove il runtime di Node.js carica la funzione di terze parti nel codice.

Funzionalità del Server

Poiché Node.js controlla il back-end delle applicazioni web, uno dei suoi compiti principali è quello di gestire le richieste HTTP.

Ecco un riassunto di come i server web gestiscono le richieste HTTP in entrata. La funzionalità del server è quella di ascoltare le richieste, determinare il più rapidamente possibile la risposta di cui ciascuno ha bisogno, e restituire tale risposta al mittente della richiesta. Questa applicazione deve ricevere una richiesta HTTP in entrata innescata dall'utente, analizzare la richiesta, eseguire il calcolo, generare la risposta e rimandarla indietro. Un modulo HTTP come Node.js viene utilizzato perché semplifica questi passaggi, permettendo al programmatore web di concentrarsi sull'applicazione.

Considera il seguente esempio che implementa questa funzionalità di base:

```

const http = require('http');
const url = require('url');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  const queryObject = url.parse(req.url, true).query;
  let result = parseInt(queryObject.a) + parseInt(queryObject.b);

  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(`Result: ${result}\n`);
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});

```

Salva questi contenuti in un file chiamato `basic_server.js` ed eseguilo attraverso un comando `node`. Il terminale che esegue Node.js mostrerà il seguente messaggio:

```
Server running at http://127.0.0.1:3000/
```

Poi visita la seguente URL nel tuo browser web: `http://127.0.0.1:3000/numbers?a=2&b=17`

Node.js esegue un server web nel tuo computer e usa due moduli: `http` e `url`. Il modulo `http` imposta un server HTTP di base, elabora le richieste web in arrivo e le passa al nostro semplice codice applicativo. Il modulo `URL` analizza gli argomenti passati nella URL, li converte in un formato intero ed esegue l'operazione di addizione. Il modulo `http` invia quindi la risposta come testo al browser web.

In una vera applicazione web, Node.js è comunemente usato per elaborare e recuperare dati, di solito da un database, e restituire le informazioni elaborate al front-end per visualizzarle. Ma l'applicazione di base in questa lezione mostra concisamente come Node.js faccia uso di moduli per gestire le richieste web come un server web.

Esercizi Guidati

1. Quali sono le ragioni per usare moduli invece di scrivere semplici funzioni?

2. Perché l'ambiente Node.js è diventato così popolare? Citane una caratteristica.

3. Qual è lo scopo del file `package.json`?

4. Perché non è raccomandato salvare e condividere la cartella `node_modules`?

Esercizi Esplorativi

1. Come puoi eseguire applicazioni Node.js sul tuo computer?

2. Come si possono delimitare i parametri da analizzare nella URL all'interno del server?

3. Indica uno scenario in cui un compito specifico potrebbe essere un collo di bottiglia per un'applicazione Node.js.

4. Come implementeresti un parametro per moltiplicare o sommare i due numeri nell'esempio del server?

Sommario

Questa lezione fornisce una panoramica dell'ambiente Node.js, le sue caratteristiche e come può essere usato per implementare semplici programmi. Questa lezione include i seguenti concetti:

- Cos'è Node.js e perché viene usato.
- Come eseguire programmi Node.js usando la linea di comando.
- I cicli di eventi e il singolo thread.
- I moduli.
- Node Package Manager (NPM).
- Funzionalità del server.

Risposte agli Esercizi Guidati

1. Quali sono le ragioni per usare moduli invece di scrivere semplici funzioni?

Optando per i moduli invece delle funzioni convenzionali, il programmatore crea una base di codice più semplice da leggere e mantenere e per la quale scrivere test automatici.

2. Perché l'ambiente Node.js è diventato così popolare? Citane una caratteristica.

Una ragione è la flessibilità del linguaggio JavaScript, che era già ampiamente utilizzato nel front-end delle applicazioni web. Node.js permette l'uso di un solo linguaggio di programmazione in tutto il sistema.

3. Qual è lo scopo del file `package.json`?

Questo file contiene metadati per il progetto, come il nome, la versione, le dipendenze (librerie) e così via. Dato un certo file `package.json`, altre persone possono scaricare e installare le stesse librerie ed eseguire test nello stesso modo in cui lo ha fatto il creatore originale.

4. Perché non è raccomandato salvare e condividere la cartella `node_modules`?

La cartella `node_modules` contiene le implementazioni delle librerie disponibili in repository remoti. Quindi il modo migliore per condividere queste librerie è quello di indicarle nel file `package.json` e poi usare NPM per scaricare queste librerie. Questo metodo è più semplice e privo di errori, perché non è necessario rintracciare e mantenere le librerie localmente.

Risposte agli Esercizi Esplorativi

1. Come puoi eseguire applicazioni Node.js sul tuo computer?

Puoi eseguirle digitando `node PATH/FILE_NAME.js` nella linea di comando nel tuo terminale, cambiando PATH con il percorso del tuo file Node.js e cambiando FILE_NAME.js con il nome del file scelto.

2. Come si possono delimitare i parametri da analizzare nella URL all'interno del server?

Il carattere *ampersand* & è usato per delimitare questi parametri, in modo che possano essere estratti e analizzati nel codice JavaScript.

3. Indica uno scenario in cui un compito specifico potrebbe essere un collo di bottiglia per un'applicazione Node.js.

Node.js non è un buon ambiente in cui eseguire processi intensivi per la CPU perché usa un singolo thread. Uno scenario di calcolo numerico potrebbe rallentare e bloccare l'intera applicazione. Se è necessaria una simulazione numerica, è meglio usare altri strumenti.

4. Come implementeresti un parametro per moltiplicare o sommare i due numeri nell'esempio del server?

Usa un operatore ternario o una condizione *if-else* per controllare un parametro aggiuntivo. Se il parametro è la stringa `mult`, questa restituisce il prodotto dei numeri, altrimenti restituisce la somma. Sostituisci il vecchio codice con quello qui sotto. Riavvia il server dalla linea di comando premendo `Ctrl + C` e rilanciando il comando per riavviare il server. Ora prova la nuova applicazione visitando l'URL `http://127.0.0.1:3000/numbers?a=2&b=17&operation=mult` nel tuo browser. Se ometti o cambi l'ultimo parametro, il risultato dovrebbe essere la somma dei numeri.

```
let result = queryObject.operation == 'mult' ? parseInt(queryObject.a) *  
parseInt(queryObject.b) : parseInt(queryObject.a) + parseInt(queryObject.b);
```



035.2 Fondamenti di Node.js Express

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 035.2

Peso

4

Arese di Conoscenza Chiave

- Definire percorsi verso file statici e modelli EJS
- Erogare file statici attraverso Express
- Erogare i modelli EJS attraverso Express
- Creare semplici modelli EJS non annidati
- Utilizzare l'oggetto request per accedere ai parametri HTTP GET e POST ed elaborare i dati inviati attraverso i moduli HTML
- Conoscenza della convalida dell'input dell'utente
- Conoscenza del cross-site scripting (XSS)
- Conoscenza del cross-site request forgery (CSRF)

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- Express e body-parser node module
- Express app object
- `app.get()`, `app.post()`
- `res.query`, `res.body`
- `ejs` node module

- `res.render()`
- `<% ... %>, <%= ... %>, <%# ... %>, <%- ... %>`
- `views/`



035.2 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	035 Programmazione Server con Node.js
Obiettivo:	035.2 Fondamenti di Node.js Express
Lezione:	1 di 2

Introduzione

Express.js, o semplicemente Express, è un popolare framework che gira su Node.js ed è usato per realizzare server HTTP che gestiscono le richieste dei client delle applicazioni web. Express supporta molti modi per leggere i parametri inviati via HTTP.

Script di Inizializzazione Server

Per dimostrare le caratteristiche di base di Express nel ricevere e gestire le richieste, simuliamo un'applicazione che richiede alcune informazioni dal server. In particolare, il server di esempio:

- Fornisce una funzione echo, che restituisce semplicemente il messaggio inviato dal client.
- Comunica al client il suo indirizzo IP su richiesta.
- Usa i cookie per identificare i client conosciuti.

Il primo passo è creare il file JavaScript che opererà come server. Usando `npm`, create una directory chiamata `myserver` con il file JavaScript:

```
$ mkdir myserver
$ cd myserver/
$ npm init
```

Per il file di inizializzazione, può essere usato qualsiasi nome di file. Qui useremo il nome predefinito del file: `index.js`. Il seguente elenco mostra un file di base `index.js` che sarà usato come punto di ingresso per il nostro server:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.get('/', (req, res) => {
  res.send('Request received')
})

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

Alcune costanti importanti per la configurazione del server sono definite nelle prime righe dello script. Le prime due, `express` e `app`, corrispondono al modulo incluso `express` e a un'istanza di questo modulo che esegue la nostra applicazione. All'oggetto `app` aggiungeremo le azioni che devono essere eseguite dal server.

Le altre due costanti, `host` e `port`, definiscono l'host e la porta di comunicazione associati al server.

Se hai un host accessibile pubblicamente, usa il suo nome invece di `myserver` come valore di `host`. Se non fornisci il nome dell'host, Express utilizzerà di default `localhost`, ovvero il computer su cui l'applicazione viene eseguita. In questo caso, nessun client esterno sarà in grado di raggiungere il programma, il che può andare bene per i test ma offre poca utilità in produzione.

La porta deve essere fornita, altrimenti il server non partirà.

Questo script allega solo due procedure all'oggetto `app`: l'azione `app.get()` che risponde alle richieste fatte dai client tramite HTTP GET, e la chiamata `app.listen()`, che è necessaria per attivare il server e gli assegna un host e una porta.

Per avviare il server, basta eseguire il comando `node`, fornendo il nome dello script come argomento:

```
$ node index.js
```

Non appena appare il messaggio `Server ready at http://myserver:8080`, il server è pronto a ricevere richieste da un client HTTP. Le richieste possono essere fatte da un browser sullo stesso computer su cui il server è in esecuzione, o da un'altra macchina che può accedere al server.

Tutti i dettagli della transazione che vedremo qui sono mostrati nel browser se si apre la console dello sviluppatore. In alternativa, il comando `curl` può essere usato per la comunicazione HTTP e permette di ispezionare più facilmente i dettagli della connessione. Se non si ha familiarità con la riga di comando della shell, si può creare un modulo HTML per inviare richieste a un server.

L'esempio seguente mostra come usare il comando `curl` sulla linea di comando per fare una richiesta HTTP al server appena attivato:

```
$ curl http://myserver:8080 -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
> Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/html; charset=utf-8
< Content-Length: 16
< ETag: W/"10-1WVvDtVyAF0vX9evlsFlfiJTT5c"
< Date: Fri, 02 Jul 2021 14:35:11 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Request received
```

L'opzione `-v` del comando `curl` visualizza tutte le intestazioni di richiesta e di risposta, così come altre informazioni di debug. Le linee che iniziano con `>` indicano gli header di richiesta inviati dal client e le linee che iniziano con `<` indicano gli header di risposta inviati dal server. Le linee che iniziano con `*` sono informazioni generate da `curl` stesso. Il contenuto della risposta viene mostrato solo alla fine, che in questo caso è la linea `Request received`.

L'URL del servizio, che in questo caso contiene l'hostname e la porta del server (`http://myserver:8080`), è stata data come argomento al comando `curl`. Non essendo stata fornita alcuna directory o nome di file, la sessione inizierà facendo riferimento alla directory principale `/`. Lo slash compare come file di richiesta nella linea `> GET / HTTP/1.1`, che è seguita nell'output dall'hostname e dalla porta.

Oltre a visualizzare le intestazioni di connessione HTTP, il comando `curl` facilita lo sviluppo dell'applicazione permettendoti di inviare dati al server usando diversi metodi HTTP e in diversi formati. Questa flessibilità rende più facile il debug di qualsiasi problema e l'implementazione di nuove funzionalità sul server.

Rotte

Le richieste che il client può fare al server dipendono da quali *route* sono state definite nel file `index.js`. Una rotta indica un metodo HTTP e definisce un *percorso* (più precisamente, una URI) che può essere richiesta dal client.

Finora, il server ha solo una rotta configurata:

```
app.get('/', (req, res) => {
  res.send('Request received')
})
```

Anche se si tratta di una rotta molto semplice, che restituisce semplicemente un messaggio di testo al client, è sufficiente per identificare i componenti più importanti che vengono utilizzati per strutturare la maggior parte delle rotte:

- Il metodo HTTP servito dalla rotta. Nell'esempio, il metodo HTTP `GET` è indicato dalla proprietà `get` dell'oggetto `app`.
- Il percorso servito da una rotta. Quando il client non specifica un percorso per la richiesta, il server utilizza la directory principale, che è la directory di base riservata all'utilizzo da parte del server web. Un esempio successivo in questo capitolo usa il percorso `/echo`, che corrisponde a una richiesta fatta a `myserver:8080/echo`.
- La funzione eseguita quando il server riceve una richiesta su questa rotta, di solito scritta utilizzando la forma abbreviata `=>` di una *arrow function*. Il parametro `req` (abbreviazione di “request”) e il parametro `res` (abbreviazione di “response”) forniscono dettagli sulla connessione, passati alla funzione dall'istanza dell'app stessa.

Metodo POST

Per estendere la funzionalità del nostro server di prova, vediamo come definire una rotta per il metodo HTTP POST. È usato dai client quando hanno bisogno di inviare dati extra al server oltre a quelli inclusi nell'intestazione della richiesta. L'opzione `--data` del comando `curl` invoca automaticamente il metodo POST e include il contenuto che sarà inviato al server tramite POST. La linea POST / HTTP/1.1 nel seguente output mostra che è stato usato il metodo POST. Tuttavia, il nostro server ha definito solo un metodo GET, quindi si verifica un errore quando usiamo `curl` per inviare una richiesta via POST:

```
$ curl http://myserver:8080/echo --data message="This is the POST request body"
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> POST / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
> Accept: /
> Content-Length: 37
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 37 out of 37 bytes
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< X-Powered-By: Express
< Content-Security-Policy: default-src 'none'
< X-Content-Type-Options: nosniff
< Content-Type: text/html; charset=utf-8
< Content-Length: 140
< Date: Sat, 03 Jul 2021 02:22:45 GMT
< Connection: keep-alive
<
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot POST /</pre>
</body>
</html>
* Connection #0 to host myserver left intact
```

Nell'esempio precedente, eseguire `curl` con il parametro `--data message="This is the POST request body"` equivale a inviare un modulo contenente il campo di testo chiamato `message`, riempito con `This is the POST request body`.

Poichè il server è configurato con una sola rotta per il percorso `/`, e quella rotta risponde solo al metodo HTTP GET, l'*header* di risposta sarà `HTTP/1.1 404 Not Found`. Inoltre, Express ha generato automaticamente una breve risposta HTML con l'avviso `Cannot POST`.

Avendo visto come generare una richiesta POST attraverso `curl`, scriviamo un programma Express che possa gestire con successo la richiesta.

Per prima cosa, nota che il campo `Content-Type` nell'intestazione della richiesta dice che i dati inviati dal client sono in formato `application/x-www-form-urlencoded`. Express non riconosce questo formato per default, quindi dobbiamo usare il modulo `express.urlencoded`. Quando includiamo questo modulo, l'oggetto `req`—passato come parametro alla funzione `handler`—ha la proprietà `req.body.message` impostata, che corrisponde al campo `message` inviato dal client. Il modulo viene caricato con `app.use`, che dovrebbe essere posto prima della dichiarazione delle rotte:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.use(express.urlencoded({ extended: true }))
```

Una volta fatto questo sarebbe sufficiente cambiare `app.get` in `app.post` nella rotta esistente per soddisfare le richieste fatte tramite POST e recuperare il corpo della richiesta:

```
app.post('/', (req, res) => {
  res.send(req.body.message)
})
```

Invece di sostituire la rotta, un'altra possibilità sarebbe quella di aggiungere semplicemente questa nuova rotta, perché Express identifica il metodo HTTP nell'intestazione della richiesta e usa la rotta appropriata. Poiché siamo interessati ad aggiungere più di una funzionalità a questo server, è conveniente separare ciascuna di esse con il proprio percorso, come `/echo` e `/ip`.

Path e Function Handler

Avendo definito quale metodo HTTP risponderà alla richiesta, ora abbiamo bisogno di definire un

percorso specifico per la risorsa e una funzione che elabori e generi una risposta al client.

Per espandere la funzionalità echo del server possiamo definire una rotta usando il metodo POST con il percorso /echo:

```
app.post('/echo', (req, res) => {
  res.send(req.body.message)
})
```

Il parametro `req` della funzione handler contiene tutti i dettagli della richiesta memorizzati come proprietà. Il contenuto del campo `message` nel corpo della richiesta è disponibile nella proprietà `req.body.message`. L'esempio invia semplicemente questo campo al client attraverso la chiamata `res.send(req.body.message)`.

Ricorda che i cambiamenti apportati hanno effetto solo dopo che il server è stato riavviato. Poiché stai eseguendo il server da una finestra di terminale durante gli esempi di questo capitolo, puoi spegnere il server premendo `Ctrl + C` sul terminale. Poi rilancia il server attraverso il comando `node index.js`. La risposta ottenuta dal client alla richiesta `curl` che abbiamo mostrato prima avrà ora successo:

```
$ curl http://myserver:8080/echo --data message="This is the POST request body"
This is the POST request body
```

Altri Modi per Passare e Restituire Informazioni in una Richiesta GET

Potrebbe essere eccessivo usare il metodo HTTP POST per inviare solo brevi messaggi di testo come quello usato nell'esempio. In questi casi, i dati possono essere inviati in una *query string* che inizia con un punto interrogativo. Così facendo, la stringa `?message=This+is+the+message` potrebbe essere inclusa nel percorso di richiesta del metodo HTTP GET. I campi usati nella stringa di richiesta sono disponibili al server nella proprietà `req.query`. Pertanto, un campo chiamato `message` è disponibile nella proprietà `req.query.message`.

Un altro modo per inviare dati tramite il metodo HTTP GET è quello di usare i *parametri di rotta* di Express:

```
app.get('/echo/:message', (req, res) => {
  res.send(req.params.message)
})
```

La rotta in questo esempio corrisponde alle richieste fatte con il metodo GET usando il percorso /echo/:message, dove :message è un marcitore per qualsiasi termine inviato con quella etichetta dal client. Questi parametri sono accessibili nella proprietà req.params. Con questo nuovo percorso, la funzione echo del server può essere richiesta più succintamente dal client:

```
$ curl http://myserver:8080/echo/hello
hello
```

In altre situazioni, le informazioni di cui il server ha bisogno per elaborare la richiesta non devono essere fornite esplicitamente dal client. Per esempio, il server ha un altro modo per recuperare l'indirizzo IP pubblico del client. Questa informazione è presente nell'oggetto req per default, nella proprietà req.ip:

```
app.get('/ip', (req, res) => {
  res.send(req.ip)
})
```

Ora il client può richiedere il percorso /ip con il metodo GET per trovare il proprio indirizzo IP pubblico:

```
$ curl http://myserver:8080/ip
187.34.178.12
```

Altre proprietà dell'oggetto req possono essere modificate dal client, specialmente le intestazioni della richiesta disponibili in req.headers. La proprietà req.headers.user-agent, per esempio, identifica quale programma stia eseguendo la richiesta. Anche se non è una pratica comune, il client può cambiare il contenuto di questo campo, quindi il server non dovrebbe usarlo per identificare in modo affidabile un particolare client. È ancora più importante validare i dati forniti esplicitamente dal client, per evitare incongruenze nei termini e nei formati che potrebbero influenzare negativamente l'applicazione.

Adeguamenti alla Risposta

Come abbiamo visto negli esempi precedenti, il parametro res è responsabile della restituzione di una risposta al client. Inoltre, l'oggetto res può cambiare altri aspetti della risposta. Avrai notato che, sebbene le risposte che abbiamo implementato finora siano solo brevi messaggi di testo semplice, l'intestazione Content-Type delle risposte usa text/html; charset=utf-8. Anche se questo non impedisce alla risposta in testo semplice di essere accettata, sarà più corretto se ridefiniamo questo campo nell'intestazione della risposta a text/plain con l'impostazione res.type('text/plain').

Altri tipi di aggiustamenti della risposta implicano l'uso di *cookie*, che permettono al server di identificare un client che ha precedentemente fatto una richiesta. I cookie sono importanti per funzioni avanzate, come la creazione di sessioni private che associano le richieste a un utente specifico, ma qui tratteremo solo un semplice esempio di come usare un cookie per identificare un client che ha precedentemente eseguito l'accesso al server.

Dato il design modulare di Express, la gestione dei cookie deve essere installata con il comando `npm` prima di essere usata nello script:

```
$ npm install cookie-parser
```

Dopo l'installazione, la gestione dei cookie deve essere inclusa nello script del server. La seguente definizione dovrebbe essere inclusa nelle prime righe del file:

```
const cookieParser = require('cookie-parser')
app.use(cookieParser())
```

Per illustrare l'uso dei cookie, modifichiamo la funzione handler della rotta con il percorso di root `/` che esiste già nello script. L'oggetto `req` ha una proprietà `req.cookies`, dove vengono conservati i cookie inviati nell'intestazione della richiesta. L'oggetto `res`, d'altra parte, ha un metodo `res.cookie()` che crea un nuovo cookie da inviare al client. La funzione *handler* nell'esempio seguente controlla se nella richiesta esiste un cookie con il nome `known`. Se tale cookie non esiste, il server assume che si tratti di un primo visitatore e gli invia un cookie con quel nome attraverso la chiamata `res.cookie('known', '1')`. Noi assegniamo arbitrariamente il valore `1` al cookie perché si suppone che abbia qualche contenuto, ma il server non consulta quel valore. Questa applicazione assume semplicemente che la semplice presenza del cookie indichi che il client ha già richiesto questo percorso in precedenza:

```
app.get('/', (req, res) => {
  res.type('text/plain')
  if ( req.cookies.known === undefined ){
    res.cookie('known', '1')
    res.send('Welcome, new visitor!')
  }
  else
    res.send('Welcome back, visitor!');
})
```

Per impostazione predefinita, `curl` non usa i cookie nelle transazioni. Ma ha opzioni per

memorizzare (`-c cookies.txt`) e inviare i cookie memorizzati (`-b cookies.txt`):

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
> Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/plain; charset=utf-8
* Added cookie known="1" for domain myserver, path /, expire 0
< Set-Cookie: known=1; Path=/
< Content-Length: 21
< ETag: W/"15-l7qrxcqicl4xv6EfA5fZFWCFrgY"
< Date: Sat, 03 Jul 2021 23:45:03 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Welcome, new visitor!
```

Poiché questo comando era il primo con cui effettuavamo un accesso dall'implementazione dei cookie, il client non ne ha fornito nessuno nella richiesta. Come previsto, quindi, il server non trovando il cookie nella richiesta ne ha incluso uno nelle intestazioni di risposta, come indicato nella linea `Set-Cookie: known=1; Path=/` dell'output. Poiché abbiamo abilitato i cookie in `curl`, una nuova richiesta includerà il cookie `known=1` nelle intestazioni della richiesta, permettendo al server di identificare la presenza del cookie:

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
> Accept: /
> Cookie: known=1
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/plain; charset=utf-8
< Content-Length: 21
< ETag: W/"15-ATq2fIQYtLMYIUpJwwpb5SjV9lw"
< Date: Sat, 03 Jul 2021 23:45:47 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Welcome back, visitor
```

Sicurezza dei Cookie

Lo sviluppatore dovrebbe essere consapevole delle potenziali vulnerabilità quando usa i cookie per identificare i client che fanno richieste. I criminali informatici possono usare tecniche come *cross-site scripting* (XSS) e *cross-site request forgery* (CSRF) per rubare i cookie da un client e quindi impersonarlo quando fa una richiesta al server. In generale, questi tipi di attacchi utilizzano campi di commento non convalidati o URL meticolosamente costruite per inserire codice JavaScript dannoso nella pagina. Quando viene eseguito da un client, questo codice può copiare i cookie validi e memorizzarli o inoltrarli a un'altra destinazione.

Perciò, specialmente nelle applicazioni professionali, è importante installare e usare funzioni Express più specializzate, conosciute come *middleware*. I moduli `express-session` o `cookie-session` forniscono un controllo più completo e sicuro sulla gestione della sessione e dei cookie. Questi componenti permettono controlli extra per evitare che i cookie vengano deviati dal loro emittente originale.

Esercizi Guidati

1. Come si può leggere il contenuto del campo `comment`, inviato all'interno di una stringa di query del metodo HTTP GET, in una funzione handler?

2. Scrivi una rotta che usi il metodo HTTP GET e il percorso `/agent` per rimandare al client il contenuto dell'intestazione `user-agent`.

3. Express.js ha una caratteristica chiamata *route parameters*, dove un percorso come `/user/:name` può essere usato per ricevere il parametro `name` inviato dal client. Come si può accedere al parametro `name` all'interno della funzione handler della rotta?

Esercizi Esplorativi

- Se il nome host di un server è `myserver`, quale rotta Express riceverebbe l'invio nel modulo sottostante?

```
<form action="/contact/feedback" method="post"> ... </form>
```

- Durante la fase di sviluppo del server, il programmatore non è in grado di leggere la proprietà `req.body`, anche dopo aver verificato che il client stia inviando correttamente il contenuto tramite il metodo HTTP POST. Qual è la probabile causa di questo problema?

- Cosa succede quando il server ha una rotta impostata sul percorso `/user/:name` e il client fa una richiesta a `/user/?`

Sommario

Questa lezione spiega come scrivere script Express per ricevere e gestire richieste HTTP. Express usa il concetto di *route* per definire le risorse disponibili ai client, il che dà grande flessibilità per costruire server per qualsiasi tipo di applicazione web. Questa lezione passa in rassegna i seguenti concetti e procedure:

- Percorsi che usano i metodi HTTP GET e HTTP POST.
- Come vengono memorizzati i dati del modulo nell'oggetto `request`.
- Come usare i parametri delle rotte.
- Personalizzare le intestazioni di risposta.
- Gestione di base dei cookie.

Risposte agli Esercizi Guidati

1. Come si può leggere il contenuto del campo `comment`, inviato all'interno di una stringa di query del metodo HTTP GET, in una funzione handler?

Il campo `comment` è disponibile nella proprietà `req.query.comment`.

2. Scrivi una rotta che usa il metodo HTTP GET e il percorso `/agent` per rimandare al client il contenuto dell'intestazione `user-agent`.

```
app.get('/agent', (req, res) => {
  res.send(req.headers.user-agent)
})
```

3. Express.js ha una caratteristica chiamata *route parameters*, dove un percorso come `/user/:name` può essere usato per ricevere il parametro `name` inviato dal client. Come si può accedere al parametro `name` all'interno della funzione handler della rotta?

Il parametro `name` è accessibile nella proprietà `req.params.name`.

Risposte agli Esercizi Esplorativi

- Se il nome host di un server è `myserver`, quale rotta Express riceverebbe l'invio nel modulo sottostante?

```
<form action="/contact/feedback" method="post"> ... </form>
```

```
app.post('/contact/feedback', (req, res) => {  
  ...  
})
```

- Durante la fase di sviluppo del server, il programmatore non è in grado di leggere la proprietà `req.body`, anche dopo aver verificato che il client stia inviando correttamente il contenuto tramite il metodo HTTP POST. Qual è la probabile causa di questo problema?

Il programmatore non ha incluso il modulo `express.urlencoded`, che permette a Express di estrarre il corpo di una richiesta.

- Cosa succede quando il server ha una rotta impostata sul percorso `/user/:name` e il client fa una richiesta a `/user/?`

Il server genererà una risposta `404 Not Found`, perché la rotta richiede che il parametro `:name` sia fornito dal client.



035.2 Lezione 2

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	035 Programmazione Server con Node.js
Obiettivo:	035.2 Fondamenti di Node.js Express
Lezione:	2 di 2

Introduzione

I server web hanno meccanismi molto versatili per produrre risposte alle richieste dei client. Per alcune richieste è sufficiente che il server web fornisca una risposta statica, non elaborata, perché la risorsa richiesta è la stessa per qualsiasi client. Per esempio, quando un client richiede un’immagine accessibile a tutti, è sufficiente che il server invii il file contenente l’immagine.

Ma quando le risposte vengono generate dinamicamente, potrebbe essere necessario strutturarle meglio delle semplici righe scritte nello script del server. In questi casi, è conveniente che il server web sia in grado di generare un documento completo, che può essere interpretato e visualizzato dal client. Nel contesto dello sviluppo di applicazioni web, i documenti HTML vengono comunemente creati come modelli e tenuti separati dallo script del server, che inserisce i dati dinamici in posizioni predeterminate nel modello appropriato e quindi invia la risposta formattata al client.

Le applicazioni Web spesso impiegano risorse sia statiche sia dinamiche. Un documento HTML, anche se è stato generato dinamicamente, può avere riferimenti a risorse statiche come file CSS e immagini. Per dimostrare in che modo Express aiuta a gestire questo tipo di domanda, configureremo un server di esempio che fornisce file statici; quindi implementeremo percorsi che generano risposte strutturate basate su modelli.

File Statici

Il primo passo è creare il file JavaScript che verrà eseguito come server. Seguiamo lo stesso schema trattato nelle lezioni precedenti per creare una semplice applicazione Express: prima creiamo una directory chiamata `server` poi installiamo i componenti di base con il comando `npm`:

```
$ mkdir server
$ cd server/
$ npm init
$ npm install express
```

Per il punto di ingresso, può essere usato qualsiasi nome di file, ma qui useremo il nome di default: `index.js`. Il seguente elenco mostra un file di base `index.js` che sarà usato come punto di partenza per il nostro server:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

Non è necessario scrivere codice esplicito per inviare un file statico. Express ha un middleware per questo scopo, chiamato `express.static`. Se il tuo server ha bisogno di inviare file statici al client, basta caricare il middleware `express.static` all'inizio dello script:

```
app.use(express.static('public'))
```

Il parametro `public` indica la directory che contiene i file che il client può richiedere. I percorsi richiesti dai client non devono includere la directory `public`, ma solo il nome del file, o il percorso del file relativo alla directory `public`. Per richiedere il file `public/layout.css`, per esempio, il client fa una richiesta a `/layout.css`.

Output Formattato

Mentre l'invio di contenuti statici è semplice, i contenuti generati dinamicamente possono variare ampiamente. Creare risposte dinamiche con messaggi brevi rende facile testare le applicazioni nelle

loro fasi iniziali di sviluppo. Per esempio, quello che segue è un percorso di test che rimanda semplicemente al client un messaggio inviato con il metodo HTTP POST. La risposta può semplicemente replicare il contenuto del messaggio in chiaro, senza alcuna formattazione:

```
app.post('/echo', (req, res) => {
  res.send(req.body.message)
})
```

Una rotta come questa è un buon esempio da usare quando si impara Express e per scopi diagnostici, dove una risposta grezza inviata con `res.send()` è sufficiente. Ma un server efficace deve essere in grado di produrre risposte più complesse. Passeremo ora a sviluppare un tipo di rotta più sofisticata.

La nostra nuova applicazione, invece di inviare solo il contenuto della richiesta corrente, mantiene una lista completa dei messaggi inviati nelle richieste precedenti da ogni client e invia la lista di ogni client quando richiesto. Una risposta che unisce tutti i messaggi è un'opzione, ma altre modalità di output formattate sono più appropriate, specialmente quando le risposte diventano più elaborate.

Per ricevere e memorizzare i messaggi del client inviati durante la sessione corrente, dobbiamo prima includere moduli extra per gestire i cookie e i dati inviati tramite il metodo HTTP POST. L'unico scopo del seguente server di esempio è quello di registrare i messaggi inviati tramite POST e visualizzare i messaggi precedentemente inviati quando il client emette una richiesta GET. Quindi ci sono due percorsi per il percorso /. Il primo percorso soddisfa le richieste fatte con il metodo HTTP POST e il secondo soddisfa le richieste fatte con il metodo HTTP GET:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.use(express.static('public'))

const cookieParser = require('cookie-parser')
app.use(cookieParser())

const { v4: uuidv4 } = require('uuid')

app.use(express.urlencoded({ extended: true }))

// Array to store messages
let messages = []

app.post('/', (req, res) => {
```

```
// Only JSON enabled requests
if ( req.headers.accept != "application/json" )
{
  res.sendStatus(404)
  return
}

// Locate cookie in the request
let uuid = req.cookies.uuid

// If there is no uuid cookie, create a new one
if ( uuid === undefined )
  uuid = uuidv4()

// Add message first in the messages array
messages.unshift({uuid: uuid, message: req.body.message})

// Collect all previous messages for uuid
let user_entries = []
messages.forEach( entry ) => {
  if ( entry.uuid == req.cookies.uuid )
    user_entries.push(entry.message)
}

// Update cookie expiration date
let expires = new Date(Date.now());
expires.setDate(expires.getDate() + 30);
res.cookie('uuid', uuid, { expires: expires })

// Send back JSON response
res.json(user_entries)

})

app.get('/', (req, res) => {

// Only JSON enabled requests
if ( req.headers.accept != "application/json" )
{
  res.sendStatus(404)
  return
}

// Locate cookie in the request
```

```

let uuid = req.cookies.uuid

// Client's own messages
let user_entries = []

// If there is no uuid cookie, create a new one
if (uuid === undefined){
    uuid = uuidv4()
}
else {
    // Collect messages for uuid
    messages.forEach( entry => {
        if (entry.uuid == req.cookies.uuid)
            user_entries.push(entry.message)
    })
}

// Update cookie expiration date
let expires = new Date(Date.now());
expires.setDate(expires.getDate() + 30);
res.cookie('uuid', uuid, { expires: expires })

// Send back JSON response
res.json(user_entries)

})

app.listen(port, host, () => {
    console.log(`Server ready at http://${host}:${port}`)
})

```

Abbiamo mantenuto la configurazione dei file statici all'inizio del file, perché sarà presto utile fornire file statici come `layout.css`. Oltre al middleware `cookie-parser` introdotto nel capitolo precedente, l'esempio include anche il middleware `uuid` per generare un numero identificativo unico passato come cookie a ogni client che invia un messaggio. Se non sono già installati nella directory del server di esempio, questi moduli possono essere installati con il comando `npm install cookie-parser uuid`.

L'array globale chiamato `messages` memorizza i messaggi inviati da tutti i client. Ogni elemento di questo array consiste in un oggetto con le proprietà `uuid` e `message`.

Ciò che è veramente nuovo in questo script è il metodo `res.json()`, usato alla fine delle due rotte per generare una risposta in formato JSON con l'array contenente i messaggi già inviati dal client:

```
// Send back JSON response
res.json(user_entries)
```

JSON è un formato di testo semplice che permette di raggruppare un insieme di dati in una singola struttura associativa: il contenuto è espresso come *chiavi* e *valori*. JSON è particolarmente utile quando le risposte devono essere elaborate dal client. Usando questo formato, un oggetto o un array JavaScript può essere facilmente ricostruito sul lato client con tutte le proprietà e gli indici dell'oggetto originale sul server.

Poiché stiamo strutturando ogni messaggio in JSON, rifiutiamo le richieste che non contengono `application/json` nella loro intestazione `accept`:

```
// Only JSON enabled requests
if ( req.headers.accept != "application/json" )
{
  res.sendStatus(404)
  return
}
```

Una richiesta fatta con un semplice comando `curl` per inserire un nuovo messaggio non sarà accettata, perché `curl` per default non specifica `application/json` nell'intestazione `accept`:

```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt
Not Found
```

L'opzione `-H "accept: application/json"` cambia l'intestazione della richiesta per specificare il formato della risposta, che questa volta sarà accettata e risponderà nel formato specificato:

```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt -H "accept: application/json"
["My first message"]
```

Ottener messaggi usando l'altro percorso è fatto in modo simile, ma questa volta usando il metodo HTTP GET:

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -H "accept: application/json"
["Another message", "My first message"]
```

I Template

Le risposte in formati come JSON sono convenienti per comunicare tra i programmi, ma lo scopo principale della maggior parte dei server di applicazioni web è quello di produrre contenuti HTML per l'utilizzo umano. Incorporare codice HTML all'interno di codice JavaScript non è una buona idea, perché mischiare i linguaggi nello stesso file rende il programma più suscettibile di errori e complica la manutenzione del codice.

Express può lavorare con diversi *template engine* che separano l'HTML per il contenuto dinamico; l'elenco completo può essere trovato sul sito [Express template engines site](#). Uno dei motori di template più popolari è *Embedded JavaScript* (EJS), che permette di creare file HTML con tag specifici per l'inserimento di contenuto dinamico.

Come altri componenti di Express, EJS deve essere installato nella directory dove il server è in esecuzione:

```
$ npm install ejs
```

Successivamente, il motore EJS deve essere impostato come visualizzatore predefinito nello script del server (vicino all'inizio del file `index.js`, prima delle definizioni delle rotte):

```
app.set('view engine', 'ejs')
```

La risposta generata con il template viene inviata al client con la funzione `res.render()`, che riceve come parametri il nome del file del template e un oggetto contenente valori che saranno accessibili dall'interno del template stesso. Le rotte usate nell'esempio precedente possono essere riscritte per generare sia risposte HTML sia JSON:

```
app.post('/', (req, res) => {
  let uuid = req.cookies.uuid

  if (uuid === undefined)
    uuid = uuidv4()
```

```

messages.unshift({uuid: uuid, message: req.body.message})

let user_entries = []
messages.forEach( entry => {
  if ( entry.uuid == req.cookies.uuid )
    user_entries.push(entry.message)
})

let expires = new Date(Date.now());
expires.setDate(expires.getDate() + 30);
res.cookie('uuid', uuid, { expires: expires })

if ( req.headers.accept == "application/json" )
  res.json(user_entries)
else
  res.render('index', {title: "My messages", messages: user_entries})

})

app.get('/', (req, res) => {

let uuid = req.cookies.uuid

let user_entries = []

if ( uuid === undefined ){
  uuid = uuidv4()
}
else {
  messages.forEach( entry => {
    if ( entry.uuid == req.cookies.uuid )
      user_entries.push(entry.message)
  })
}

let expires = new Date(Date.now());
expires.setDate(expires.getDate() + 30);
res.cookie('uuid', uuid, { expires: expires })

if ( req.headers.accept == "application/json" )
  res.json(user_entries)
else
  res.render('index', {title: "My messages", messages: user_entries})

})

```

Si noti che il formato della risposta dipende dall'intestazione accept trovata nella richiesta:

```
if ( req.headers.accept == "application/json" )
  res.json(user_entries)
else
  res.render('index', {title: "My messages", messages: user_entries})
```

Una risposta in formato JSON viene inviata solo se il client la richiede esplicitamente. Altrimenti, la risposta è generata dal template `index`. Lo stesso array `user_entries` alimenta sia l'output JSON che il template, ma l'oggetto usato come parametro per quest'ultimo ha anche la proprietà `title: "My messages"`, che sarà usato come titolo all'interno del template.

Template HTML

Come i file statici, i file che contengono i template HTML risiedono nella loro propria directory. Per impostazione predefinita, EJS assume che i file dei template siano nella directory `views/`. Nell'esempio è stato usato un template chiamato `index`, quindi EJS cerca il file `views/index.ejs`. Il seguente elenco è il contenuto di un semplice template `views/index.ejs` che può essere usato con il codice di esempio:

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title><%= title %></title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="/layout.css">
</head>
<body>

<div id="interface">

<form action="/" method="post">
<p>
  <input type="text" name="message">
  <input type="submit" value="Submit">
</p>
</form>

<ul>
<% messages.forEach( message ) => { %>
<li><%= message %></li>
<% } %>
</ul>

</div>

</body>
</html>

```

Il primo tag speciale EJS è l'elemento `<title>` nella sezione `<head>`:

```
<%= title %>
```

Durante il processo di visualizzazione questo tag speciale sarà sostituito dal valore della proprietà `title` dell'oggetto passato come parametro alla funzione `res.render()`.

La maggior parte del template è costituita da codice HTML convenzionale, quindi il template contiene il modulo HTML per inviare nuovi messaggi. Il server di test risponde ai metodi HTTP GET e POST per lo stesso percorso `/`, da cui gli attributi `action="/" e method="post"` nel tag del modulo.

Altre parti del modello sono una combinazione di codice HTML e tag EJS. EJS ha tag per scopi

specifici all'interno del modello:

`<% ... %>`

Inserisce il controllo del flusso. Nessun contenuto è inserito direttamente da questo tag, ma può essere usato con strutture JavaScript per scegliere, ripetere o sopprimere sezioni di HTML. Esempio di avvio di un ciclo: `<% messages.forEach(message) => { %>`

`<%# ... %>`

Definisce un commento, il cui contenuto viene ignorato dal *parser*. A differenza dei commenti scritti in HTML, questi commenti non sono visibili al client.

`<%= ... %>`

Inserisce il contenuto sotto *escape* della variabile. È importante fare l'escape del contenuto sconosciuto per evitare l'esecuzione di codice JavaScript, che può aprire scappatoie per attacchi di cross-site scripting (XSS). Esempio: `<%= title %>`

`<%- ... %>`

Inserisce il contenuto della variabile senza escape.

Il mix di codice HTML e tag EJS è evidente nelle seguenti righe di codice dove i messaggi del client sono resi come una lista HTML:

```
<ul>
<% messages.forEach( message ) => { %>
<li><%= message %></li>
<% } %>
</ul>
```

Il primo tag `<% ... %>` inizia un'istruzione `forEach` che scorre tutti gli elementi dell'array `message`. I delimitatori `<%` e `%>` ti permettono di controllare le parti di codice HTML. Un nuovo elemento della lista HTML, `<%= message %>`, sarà prodotto per ogni elemento di `messages`. Con queste modifiche, il server invierà la risposta in HTML quando viene ricevuta una richiesta come la seguente:

```
$ curl http://myserver:8080/ --data message="This time" -c cookies.txt -b cookies.txt
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My messages</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="/layout.css">
</head>
<body>

<div id="interface">

<form action="/" method="post">
<p>
  <input type="text" name="message">
  <input type="submit" value="Submit">
</p>
</form>

<ul>
  <li>This time</li>
  <li>in HTML</li>
</ul>

</div>

</body>
</html>
```

La separazione tra il codice per elaborare le richieste e il codice per presentare la risposta rende il codice più pulito e permette a un team di dividere lo sviluppo dell'applicazione tra persone con specialità distinte. Un web designer, per esempio, può concentrarsi sui file template in `views/` e sui relativi fogli di stile, che sono forniti come file statici memorizzati nella directory `public/` del server di esempio.

Esercizi Guidati

1. Come dovrebbe essere configurato `express.static` in modo che i client possano richiedere i file nella directory `assets`?

2. Come si può identificare il tipo di risposta, che è specificato nell'intestazione della richiesta, in una rotta Express?

3. Quale metodo del parametro della rotta `res` (response) genera una risposta in formato JSON da un array JavaScript chiamato `content`?

Esercizi Esplorativi

1. Per impostazione predefinita, i file dei template Express si trovano nella directory `views`. Come si può modificare questa impostazione in modo che i file dei template siano memorizzati in `templates`?

2. Supponiamo che un client riceva una risposta HTML senza titolo (cioè `<title></title>`). Dopo aver verificato il template EJS, lo sviluppatore trova il tag `<title><% title %></title>` nella sezione head del file. Qual è la probabile causa del problema?

3. Usa i tag template EJS per scrivere un tag HTML `<h2></h2>` con il contenuto della variabile JavaScript `h2`. Questo tag dovrebbe essere mostrato solo se la variabile `h2` non è vuota.

Sommario

Questa lezione tratta dei metodi di base che Express.js fornisce per generare risposte strutturate e formattate ma dinamiche. È richiesto poco sforzo per impostare un server HTTP per i file statici e il sistema di template di EJS fornisce un modo semplice per generare contenuto dinamico da file HTML. Questa lezione tratta i seguenti concetti e procedure:

- Usare `express.static` per le risposte ai file statici.
- Come creare una risposta che corrisponda al campo del tipo di contenuto nell'intestazione della richiesta.
- Risposte strutturate in JSON.
- Usare i tag EJS nei template basati su HTML.

Risposte agli Esercizi Guidati

1. Come dovrebbe essere configurato `express.static` in modo che i client possano richiedere i file nella directory `assets`?

Una richiesta a `app.use(express.static('assets'))` dovrebbe essere aggiunta allo script del server.

2. Come si può identificare il tipo di risposta, che è specificato nell'intestazione della richiesta, in una rotta Express?

Il client imposta i tipi accettabili nel campo dell'intestazione `accept`, che è mappato sulla proprietà `req.headers.accept`.

3. Quale metodo del parametro della rotta `res` (response) genera una risposta in formato JSON da un array JavaScript chiamato `content`?

Il metodo `res.json(): res.json(content)`.

Risposte agli Esercizi Esplorativi

- Per impostazione predefinita, i file dei template Express si trovano nella directory `views`. Come si può modificare questa impostazione in modo che i file dei template siano memorizzati in `templates`?

La directory può essere definita nelle impostazioni iniziali dello script con `app.set('views', './templates')`.

- Supponiamo che un client riceva una risposta HTML senza titolo (cioè `<title></title>`). Dopo aver verificato il template EJS, lo sviluppatore trova il tag `<title><% title %></title>` nella sezione `head` del file. Qual è la probabile causa del problema?

+ Il tag `<%= %>` dovrebbe essere usato per racchiudere il contenuto di una variabile, come in `<%= title %>`.

- Usa i tag template EJS per scrivere un tag HTML `<h2></h2>` con il contenuto della variabile JavaScript `h2`. Questo tag dovrebbe essere mostrato solo se la variabile `h2` non è vuota.

```
<% if ( h2 != "" ) { %>
<h2><%= h2 %></h2>
<% } %>
```



035.3 Fondamenti di SQL

Obiettivi LPI di riferimento

Web Development Essentials version 1.0, Exam 030, Objective 035.3

Peso

3

Area di Conoscenza Chiave

- Stabilire una connessione al database da NodeJS
- Recuperare dati dal database in NodeJS
- Eseguire query SQL da NodeJS
- Creare semplici query SQL escludendo le join
- Comprendere le chiavi primarie
- Comprendere le variabili usate nelle query SQL
- Comprensione delle SQL injection

Di seguito è riportato un elenco parziale dei file, dei termini e dei comandi utilizzati

- sqlite3 NPM module
- Database.run(), Database.close(), Database.all(), Database.get(), Database.each()
- CREATE TABLE
- INSERT, SELECT, DELETE, UPDATE



035.3 Lezione 1

Certificazione:	Web Development Essentials
Versione:	1.0
Argomento:	035 Programmazione Server con Node.js
Obiettivo:	035.3 Fondamenti di SQL
Lezione:	1 di 1

Introduzione

Anche se è possibile scrivere le proprie funzioni per implementare la memorizzazione persistente, può essere più conveniente usare un sistema di gestione di database per accelerare lo sviluppo e assicurare una migliore sicurezza e stabilità per i dati in formato tabella. La strategia più popolare per memorizzare i dati organizzati in tabelle correlate, specialmente quando queste tabelle sono pesantemente interrogate e aggiornate, è quella di installare un database relazionale che supporti lo *Structured Query Language* (SQL), un linguaggio orientato ai database relazionali. Node.js supporta diversi sistemi di gestione di database SQL. Seguendo i principi di portabilità ed esecuzione nello spazio utente adottati da Node.js Express, SQLite è una scelta appropriata per la memorizzazione.

SQL

Lo Structured Query Language (SQL) è specifico per i database. Le operazioni di scrittura e di lettura sono espresse in frasi chiamate *statement* e *query*. Sia gli statement sia le query sono costituiti da *clausole*, che definiscono le condizioni di esecuzione dell'operazione.

Nomi e indirizzi e-mail, per esempio, possono essere memorizzati in una tabella del database che contiene i campi nome e email. Un database può contenere diverse tabelle, quindi ogni tabella deve

avere un nome unico. Se usiamo il nome `contacts` per la tabella dei nomi e delle email, un nuovo record può essere inserito con la seguente *dichiarazione* (statement):

```
INSERT INTO contacts (name, email) VALUES ("Carol", "carol@example.com");
```

Questa dichiarazione di inserimento è composta dalla clausola `INSERT INTO`, che definisce la tabella e i campi in cui i dati saranno inseriti. La seconda clausola, `VALUES`, imposta i valori che saranno inseriti. Non è necessario scrivere in maiuscolo le clausole, ma è una pratica comune, in modo da riconoscere meglio le parole chiave SQL all'interno di una dichiarazione o di una query.

Una query sulla tabella `contacts` è fatta in modo simile, ma usando la clausola `SELECT`:

```
SELECT email FROM contacts;  
dave@example.com  
carol@example.com
```

In questo caso, la clausola `SELECT email` seleziona un campo dalle voci della tabella `contacts`. La clausola `WHERE` restringe la query a righe specifiche:

```
SELECT email FROM contacts WHERE name = "Dave";  
dave@example.com
```

SQL ha molte altre clausole, e ne vedremo alcune nelle sezioni successive. Ma prima è necessario vedere come integrare il database SQL con Node.js.

SQLite

SQLite è probabilmente la soluzione più semplice per incorporare le caratteristiche del database SQL in un'applicazione. A differenza di altri popolari sistemi di gestione di database, SQLite non è un server di database a cui un client si connette. SQLite fornisce un insieme di funzioni che permettono allo sviluppatore di creare un database come un file convenzionale. Nel caso di un server HTTP implementato con Node.js Express, questo file si trova solitamente nella stessa directory dello script del server.

Prima di usare SQLite in Node.js, è necessario installare il modulo `sqlite3`. Esegui il seguente comando nella directory di installazione del server; cioè la directory che contiene lo script Node.js che eseguirai.

```
$ npm install sqlite3
```

Sai consapevole che ci sono diversi moduli che supportano SQLite, come `better-sqlite3`, il cui uso è leggermente diverso da `sqlite3`. Gli esempi in questa lezione sono per il modulo `sqlite3`, quindi potrebbero *non* funzionare come previsto se scegliete un altro modulo.

Accedere al Database

Per dimostrare come un server Node.js Express può lavorare con un database SQL, scriviamo uno script che memorizza e visualizza i messaggi inviati da un client identificato da un cookie. I messaggi sono inviati dal client tramite il metodo HTTP POST e la risposta del server può essere formattata come JSON o HTML (da un template) a seconda del formato richiesto dal client. Questa lezione non entrerà nel dettaglio dell'uso dei metodi HTTP, dei cookie e dei template. Le parti di codice mostrate qui presuppongono che tu abbia già un server Node.js Express dove queste caratteristiche sono configurate e disponibili.

Il modo più semplice per memorizzare i messaggi inviati dal client è quello di memorizzarli in un array globale, dove ogni messaggio precedentemente inviato è associato a una chiave di identificazione unica per ogni client. Questa chiave può essere inviata al client come un cookie, che viene presentato al server per recuperare i suoi messaggi precedenti nelle richieste future .

Tuttavia, questo approccio ha una debolezza: poiché i messaggi sono memorizzati solo in un array globale, tutti i messaggi saranno persi quando la sessione corrente del server verrà terminata. Questo è uno dei vantaggi di lavorare con i database, perché i dati sono memorizzati in modo persistente e non vengono persi se il server viene riavviato.

Usando il file `index.js` come script principale del server possiamo incorporare il modulo `sqlite3` e indicare il file che serve come database nella seguente maniera:

```
const sqlite3 = require('sqlite3')
const db = new sqlite3.Database('messages.sqlite3');
```

Se non esiste già, il file `messages.sqlite3` verrà creato nella stessa directory del file `index.js`. All'interno di questo singolo file, saranno memorizzate tutte le strutture e i rispettivi dati. Tutte le operazioni sul database eseguite nello script saranno intermediate dalla costante `db`, che è il nome dato al nuovo oggetto `sqlite3` che apre il file `messages.sqlite3`.

Struttura di una Tabella

Nessun dato può essere inserito nel database finché non viene creata almeno una tabella. Le tabelle vengono create con l'istruzione CREATE TABLE:

```
db.run('CREATE TABLE IF NOT EXISTS messages (id INTEGER PRIMARY KEY AUTOINCREMENT,
    uuid CHAR(36), message TEXT)')
```

Il metodo `db.run()` è usato per eseguire istruzioni SQL nel database. L'istruzione stessa è scritta come parametro per il metodo. Anche se le istruzioni SQL devono terminare con un punto e virgola quando sono inserite in un programma a riga di comando, il punto e virgola è opzionale nelle istruzioni passate come parametri in un programma.

Poiché il metodo `run` verrà eseguito ogni volta che lo script viene lanciato con `node index.js`, l'istruzione SQL include la clausola condizionale `IF NOT EXISTS` per evitare errori nelle esecuzioni future quando la tabella `messages` esisterà già.

I campi che compongono la tabella `messages` sono `id`, `uuid` e `message`. Il campo `id` è un intero unico usato per identificare ogni voce nella tabella, quindi è creato come `PRIMARY KEY`. Le chiavi primarie *non* possono essere nulle e *non* ce ne possono essere due identiche nella stessa tabella. Pertanto, quasi ogni tabella SQL ha una chiave primaria per tracciare il contenuto della tabella. Anche se è possibile scegliere esplicitamente il valore per la chiave primaria di un nuovo record (purché non esista ancora nella tabella), è conveniente che la chiave sia generata automaticamente. Il flag `AUTOINCREMENT` nel campo `id` è usato a questo scopo.

NOTE

L'impostazione esplicita delle chiavi primarie in SQLite è opzionale, perché SQLite stesso crea automaticamente una chiave primaria. Come indicato nella documentazione di SQLite: “In SQLite, le righe di una tabella hanno normalmente un intero firmato a 64 bit ROWID che è unico tra tutte le righe della stessa tabella. Se una tabella contiene una colonna di tipo `INTEGER PRIMARY KEY`, allora quella colonna diventa un alias per il ROWID. Puoi quindi accedere al ROWID usando uno qualsiasi di quattro nomi diversi, i tre nomi originali descritti sopra, o il nome dato alla colonna `INTEGER PRIMARY KEY`. Tutti questi nomi sono alias l'uno dell'altro e funzionano ugualmente bene in qualsiasi contesto.”

I campi `uuid` e `message` memorizzano rispettivamente l'identificazione del cliente e il contenuto del messaggio. Un campo di tipo `CHAR(36)` memorizza una quantità fissa di 36 caratteri, e un campo di tipo `TEXT` memorizza testi di lunghezza arbitraria.

Inserimento dei Dati

La funzione principale del nostro server di esempio è quella di memorizzare i messaggi che sono collegati al client che li ha inviati. Il client invia il messaggio nel campo `message` all'interno del corpo della richiesta inviata con il metodo HTTP POST. L'identificazione del client è in un cookie chiamato `uuid`. Con queste informazioni, possiamo scrivere quanto necessario per inserire nuovi messaggi nel database:

```
app.post('/', (req, res) => {

  let uuid = req.cookies.uuid

  if (uuid === undefined)
    uuid = uuidv4()

  // Insert new message into the database
  db.run('INSERT INTO messages (uuid, message) VALUES (?, ?)', uuid, req.body.message)

  // If an error occurs, err object contains the error message.
  db.all('SELECT id, message FROM messages WHERE uuid = ?', uuid, (err, rows) => {

    let expires = new Date(Date.now());
    expires.setDate(expires.getDate() + 30);
    res.cookie('uuid', uuid, { expires: expires })

    if (req.headers.accept == "application/json")
      res.json(rows)
    else
      res.render('index', {title: "My messages", rows: rows})

  })
})
```

Questa volta il metodo `db.run()` esegue una dichiarazione di inserimento, ma si noti che sia `uuid` sia `req.body.message` non sono scritti direttamente nella linea di dichiarazione. Invece, i punti interrogativi sono stati sostituiti per i valori. Ogni punto interrogativo corrisponde ad un parametro che segue lo *statement SQL* nel metodo `db.run()`.

Usare i punti interrogativi come marcatore nell'istruzione che viene eseguita nel database rende più facile per SQLite distinguere tra gli elementi statici dell'istruzione e i suoi dati variabili. Questa strategia permette a SQLite di *eliminare* o *pulire* il contenuto delle variabili che sono parte dello

statement, prevenendo una comune violazione della sicurezza chiamata *SQL injection*. In questo attacco, utenti malintenzionati inseriscono dichiarazioni SQL nei dati variabili nella speranza che le dichiarazioni vengano eseguite inavvertitamente: la “pulizia” sventa l’attacco disabilitando i caratteri pericolosi nei dati.

Query

Come mostrato nel codice di esempio, il nostro intento è quello di utilizzare lo stesso metodo per inserire nuovi messaggi nel database e per generare la lista dei messaggi precedentemente inviati. Il metodo `db.all()` restituisce l’insieme di tutte le voci della tabella che corrispondono ai criteri definiti nella query.

A differenza delle istruzioni eseguite da `db.run()`, `db.all()` genera una lista di record che sono gestiti dalla funzione arrow (\Rightarrow) designata nell’ultimo parametro:

```
(err, rows) => {}
```

Questa funzione, a sua volta, prende due parametri: `err` e `rows`. Il parametro `err` viene utilizzato se si verifica un errore che impedisce l’esecuzione della query. In caso di successo, tutti i record sono disponibili nell’array `rows`, dove ogni elemento è un oggetto corrispondente a un singolo record della tabella. Le proprietà di questo oggetto corrispondono ai nomi dei campi indicati nella query: `uuid` e `message`.

L’array `rows` è una struttura dati JavaScript. Come tale, può essere usato per generare risposte con metodi forniti da Node.js Express, come `res.json()` e `res.render()`. Quando viene mostrato all’interno di un template EJS, un ciclo convenzionale può elencare tutti i record:

```
<ul>
<% rows.forEach( (row) => { %>
<li><strong><%= row.id %></strong>: <%= row.message %></li>
<% } ) %>
</ul>
```

Invece di riempire l’array `rows` con tutti i record restituiti dalla query, in alcuni casi potrebbe essere più conveniente trattare ogni record individualmente con il metodo `db.each()`. La sintassi del metodo `db.each()` è simile al metodo `db.all()`, ma il parametro `row` in `(err, row) => {}` corrisponde ad un singolo record alla volta.

Modificare i Contenuti di un Database

Finora il nostro client può solo aggiungere e interrogare i messaggi sul server. Dato che il client ora conosce l' `id` dei messaggi precedentemente inviati, possiamo implementare una funzione per modificare un record specifico. Il messaggio modificato può anche essere inviato a una destinazione con metodo HTTP POST, ma questa volta con un parametro per catturare l' `id` dato dal client nel percorso della richiesta:

```
app.post('/:id', (req, res) => {
  let uuid = req.cookies.uuid

  if (uuid === undefined) {
    uuid = uuidv4()
    // 401 Unauthorized
    res.sendStatus(401)
  }
  else {

    // Update the stored message
    // using named parameters
    let param = {
      $message: req.body.message,
      $id: req.params.id,
      $uuid: uuid
    }

    db.run('UPDATE messages SET message = $message WHERE id = $id AND uuid = $uuid',
      param, function(err){

        if (this.changes > 0)
        {
          // A 204 (No Content) status code means the action has
          // been enacted and no further information is to be supplied.
          res.sendStatus(204)
        }
        else
          res.sendStatus(404)

      })
  }
})
```

Questo dimostra come usare le clausole UPDATE e WHERE per modificare un record esistente.

Un'importante differenza rispetto agli esempi precedenti è l'uso di *parametri nominali*, dove i valori sono raggruppati in un singolo oggetto (`param`) e passati al metodo `db.run()` invece di specificare ogni valore da solo. In questo caso, i nomi dei campi (preceduti da `$`) sono le proprietà dell'oggetto. I parametri con nome permettono di usare i nomi dei campi (preceduti da `$`) come marcatori invece che come punti interrogativi.

Un'istruzione come quella dell'esempio non causerà alcuna modifica al database se la condizione imposta dalla clausola `WHERE` non corrisponde a qualche record della tabella. Per valutare se qualche record è stato modificato dall'istruzione, una funzione di callback può essere usata come ultimo parametro del metodo `db.run()`. All'interno della funzione, il numero di record modificati può essere interrogato da `this.changes`. Notate che le funzioni arrow non possono essere usate in questo caso, perché solo le funzioni regolari espresse nella forma `function(){} definiscono l'oggetto this.`

La rimozione di un record è molto simile alla sua modifica. Possiamo, per esempio, continuare a usare il parametro `:id` per identificare il messaggio da cancellare, ma questa volta invocata dal metodo HTTP DELETE del client:

```
app.delete('/:id', (req, res) => {
  let uuid = req.cookies.uuid

  if (uuid === undefined){
    uuid = uuidv4()
    res.sendStatus(401)
  }
  else {
    // Named parameters
    let param = {
      $id: req.params.id,
      $uuid: uuid
    }

    db.run('DELETE FROM messages WHERE id = $id AND uuid = $uuid', param, function(err){
      if (this.changes > 0)
        res.sendStatus(204)
      else
        res.sendStatus(404)
    })
  }
})
```

I record vengono cancellati da una tabella con la clausola `DELETE FROM`. Abbiamo di nuovo usato la

funzione di callback per valutare quante voci sono state rimosse dalla tabella.

Chiudere un Database

Una volta definito, l'oggetto `db` può essere referenziato in qualsiasi momento durante l'esecuzione dello script, perché il file del database rimane aperto durante la sessione corrente. Non è comune chiudere il database mentre lo script è in esecuzione.

Una funzione per chiudere il database è utile, comunque, per evitare di chiudere bruscamente il database quando il processo del server finisce. Anche se è improbabile, chiudere bruscamente il database può causare incoerenze se i dati in-memoria non sono ancora stati trasferiti nel file. Per esempio, una chiusura improvvisa del database con perdita di dati può verificarsi se lo script viene terminato dall'utente premendo la scorciatoia da tastiera `Ctrl + C`.

Nello scenario `Ctrl + C` appena descritto, il metodo `process.on()` può intercettare i segnali inviati dal sistema operativo ed eseguire un arresto ordinato sia del database che del server:

```
process.on('SIGINT', () => {
  db.close()
  server.close()
  console.log('HTTP server closed')
})
```

La scorciatoia `Ctrl + C` invoca il segnale del sistema operativo `SIGINT`, che termina un programma in primo piano nel terminale. Prima di terminare il processo alla ricezione del segnale `SIGINT`, il sistema invoca la funzione di *callback* (l'ultimo parametro nel metodo `process.on()`). All'interno della funzione di callback, puoi mettere qualsiasi codice di pulizia, in particolare il metodo `db.close()` per chiudere il database e `server.close()`, che chiude correttamente l'istanza Node.js Express stessa.

Esercizi Guidati

1. Qual è lo scopo di una chiave primaria in una tabella di database SQL?

2. Qual è la differenza tra effettuare query usando `db.all()` e `db.each()`?

3. Perché è importante usare i mercatori e non includere i dati inviati dal client direttamente in uno statement o query SQL?

Esercizi Esplorativi

- Quale metodo nel modulo `sqlite3` può essere usato per restituire solo una voce della tabella, anche se la query corrisponde a più voci?

- Supponiamo che l'array `rows` sia stato passato come parametro a una funzione di callback e contenga il risultato di una query fatta con `db.all()`. Come può un campo chiamato `price`, che è presente nella prima posizione di `rows`, essere referenziato all'interno della funzione di callback?

- Il metodo `db.run()` esegue dichiarazioni di modifica del database, come `INSERT INTO`. Dopo aver inserito un nuovo record in una tabella, come potreste recuperare la chiave primaria del record appena inserito?

Sommario

Questa lezione spiega l'uso di base dei database SQL nelle applicazioni Node.js Express. Il modulo `sqlite3` offre un modo semplice di memorizzare dati persistenti in un database SQLite, dove un singolo file contiene l'intero database e non richiede un server di database specializzato. Questa lezione tratta i seguenti concetti e procedure:

- Come stabilire una connessione al database da Node.js.
- Come creare una semplice tabella e il ruolo delle chiavi primarie.
- Usare l'istruzione SQL `INSERT INTO` per aggiungere nuovi dati dall'interno dello script.
- Fare query SQL usando i metodi standard di SQLite e le funzioni di callback.
- Cambiare i dati nel database usando le istruzioni SQL `UPDATE` e `DELETE`.

Risposte agli Esercizi Guidati

1. Qual è lo scopo di una chiave primaria in una tabella di database SQL?

La chiave primaria è il campo di identificazione univoco per ogni record all'interno di una tabella di database.

2. Qual è la differenza tra effettuare query usando `db.all()` e `db.each()`?

Il metodo `db.all()` invoca la funzione di callback con un singolo array contenente tutte le voci corrispondenti alla query. Il metodo `db.each()` invoca la funzione di callback per ogni riga di risultato.

3. Perché è importante usare i marcatori e non includere i dati inviati dal client direttamente in uno statement o query SQL?

Con i marcatori, i dati inviati dall'utente vengono verificati prima di essere inclusi nella query o nell'istruzione. Questo ostacola gli attacchi di SQL injection, dove le istruzioni SQL sono inserite all'interno di dati variabili nel tentativo di eseguire operazioni arbitrarie sul database.

Risposte agli Esercizi Esplorativi

- Quale metodo nel modulo `sqlite3` può essere usato per restituire solo una voce della tabella, anche se la query corrisponde a più voci?

Il metodo `db.get()` ha la stessa sintassi di `db.all()`, ma restituisce solo la prima voce corrispondente alla query.

- Supponiamo che l'array `rows` sia stato passato come parametro a una funzione di callback e contenga il risultato di una query fatta con `db.all()`. Come può un campo chiamato `price`, che è presente nella prima posizione di `rows`, essere referenziato all'interno della funzione di callback?

Ogni elemento in `rows` è un oggetto le cui proprietà corrispondono ai nomi dei campi del database. Quindi il valore del campo `price` nel primo risultato è in `rows[0].price`.

- Il metodo `db.run()` esegue dichiarazioni di modifica del database, come `INSERT INTO`. Dopo aver inserito un nuovo record in una tabella, come potreste recuperare la chiave primaria del record appena inserito?

Una funzione regolare della forma `function(){}()` può essere usata come funzione di callback del metodo `db.run()`. Al suo interno, la proprietà `this.lastID` contiene il valore della chiave primaria dell'ultimo record inserito.

Imprint

© 2022 by Linux Professional Institute: Learning Materials, “Web Development Essentials (030) (Versione 1.0)”.

PDF generato: 2022-04-21

Questa opera è concessa in licenza con Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0). Per visualizzare una copia di questa licenza, visitare

<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Sebbene Linux Professional Institute si sia adoperato in buona fede per garantire che le informazioni e le istruzioni contenute in questa opera siano accurate, Linux Professional Institute declina ogni responsabilità per errori od omissioni, inclusa, senza limitazione, la responsabilità per danni derivanti dall'uso di questa opera. L'utilizzo di informazioni e istruzioni contenute in questa opera è a proprio rischio. Se qualche esempio di codice o tecnologia che questa opera contiene o descrive è soggetto a licenze open source o è sotto diritti di proprietà intellettuale di terzi, è tua responsabilità assicurarti che se ne faccia uso rispettando tali licenze e / o diritti.

I materiali didattici LPI (Learning Materials) sono un'iniziativa Linux Professional Institute (<https://lpi.org>). Materiali didattici e loro traduzioni sono su <https://learning.lpi.org>.

Per domande e commenti scrivi una mail a: learning@lpi.org.