Modularità e organizzazione del codice

Niccolò Maltoni
niccolo.maltoni@diennea.com

Perché suddividere il codice?

Le applicazioni moderne sono complesse e ricche di funzionalità.

Finora, abbiamo scritto il nostro codice JavaScript in *un solo script*, ma usare **un unico file gigante** non scala!

- Difficile orientarsi nel codice
- Conflitti di nomi creano bug subdoli
- Impossibile testare una singola funzionalità in isolamento
- Manutenzione complessa
- Lavorando in team, operare sullo stesso file genera conflitti

Nasce così la necessità di suddividere il codice in *piccole* unità *indipendenti*: i **moduli**.

Che cos'è un modulo?

Un modulo JavaScript è semplicemente un **file**; ogni script `*.js` è un modulo.

Un modulo isola il codice con una *responsabilità specifica*: ad esempio, invece di un unico `index.js`, organizziamo il codice in più file, ciascuno con uno scopo preciso:

- `validators.js` contiene funzioni per validare i dati
- `recipeService.js` contiene funzioni per comunicare con il backend
- `ui.js` contiene funzioni per rendering e UI
- `index.js` coordina e orchestra gli altri moduli

Un modulo solitamente contiene una classe o una libreria di funzioni.

I moduli sono **riusabili**: possono essere importati in più parti dell'applicazione o in progetti diversi.

Tipi di moduli in JavaScript

Come quasi tutto in JavaScript, ci sono molti modi a nostra disposizione per implementare i moduli.

Per molto tempo, infatti, JavaScript è esistito senza una vera sintassi per i moduli nel linguaggio, portando a soluzioni non standard e frammentate:

- **AMD** (***A**synchronous Module **D**efinition*): uno dei più vecchi sistemi per la gestione di moduli, inizialmente implementato dalla libreria **RequireJS**.
- **CommonJS**: il sistema per la gestione di moduli creato per **Node.js**.
- **UMD** (***U**niversal Module **D**efinition*): un altro sistema di gestione di moduli, che è stato proposto come metodo universale, compatibile sia con AMD sia con CommonJS.

Ormai tutti questi sistemi vengono lentamente abbandonati in favore di un nuovo formato standardizzato nel 2015 con ES6: gli **ECMAScript modules**.

Moduli CommonJS

CommonJS è un insieme di standard usati per implementare i moduli in JavaScript, principalmente in ambienti server-side come Node.js. Prima dell'introduzione degli ES6 Modules, CommonJS era il sistema di moduli più diffuso e ampiamente adottato.

La sintassi di CommonJS è semplice e si basa su due parole chiave principali: `require()` per importare moduli e `module.exports` per esportarli.

math.js

```
function add(a, b) {
  return a + b;
}

module.exports = { add };
```

main.js

```
const math = require('./math.js');

console.log(math.add(5, 3)); // 8
```

Di fatto, un modulo CommonJS *esporta un oggetto*, e tutte le funzioni o variabili che vogliamo rendere pubbliche devono essere aggiunte a questo oggetto.

Oggi possiamo considerare CommonJS come un sistema di moduli **legacy**: è ancora ampiamente usato in Node.js, ma non è più la scelta consigliata per nuovi progetti, vista l'esistenza di un formato di moduli "ufficiale" e standardizzato.

ECMAScript modules

Gli **ECMAScript modules** (ES Modules o ESM) sono il sistema di moduli standardizzato per JavaScript, introdotto con ES6 (ECMAScript 2015). Sono supportati nativamente in tutti i moderni browser e in Node.js (con alcune configurazioni).

La sintassi degli ES Modules è più pulita e intuitiva rispetto a CommonJS, e si basa sulle seguenti **keyword**:

- `export` per esportare elementi da un modulo
- `import` e `from` per importarli in altri moduli

math.js

```
export function add(a, b) {
  return a + b;
}

export const PI = 3.14159;
```

main.js

```
import { add, PI } from './math.js';

console.log(add(5, 3));  // 8
console.log(PI);         // 3.14159
```

ECMAScript modules: named export

La modalità che abbiamo visto nella slide precedente è chiamata **named export**. Si può utilizzare anteponendo `export` a ogni elemento che vogliamo esportare.

Possiamo esportare funzioni, variabili, classi, oggetti:

utils.js

```
// esportiamo un array
export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// esportiamo una costante
export const ES_MODULES_YEAR = 2015;

// esportiamo una classe
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

main.js

```
import { months, ES_MODULES_YEAR, User } from './utils.js'

console.log(months); // ['Jan', 'Feb', 'Mar','Apr', 'Aug',

console.log(ES_MODULES_YEAR); // 2015

const user = new User('Alice');
console.log(user.name); // "Alice"
```

Possiamo anche esportare tutto in un blocco separatamente alla fine del file:

math.js

```
function add(a, b) { return a + b; }
function subtract(a, b) { return a - b; }

export { add, subtract };
```

main.js

```
import { add, subtract } from './math.js';

console.log(add(5, 3));       // 8
console.log(subtract(5, 3));  // 2
```

ECMAScript modules: default export

Nella pratica, un modulo spesso ha **un solo elemento principale** da esportare, come ad esempio una classe. In questo caso, possiamo usare la **default export**.

user.js

```
export default class User {
  constructor(name) {
    this.name = name;
  }
}
```

main.js

```
import User from './user.js'; // non { User }, semplicemente User

new User('John');
```

In questo caso, andiamo a frapporre la keyword `default` tra `export` e la dichiarazione della funzione. Questo indica che questa è l'esportazione predefinita del modulo.

Possiamo importare la default export senza usare le parentesi graffe, e possiamo darle qualsiasi nome.

Un modulo può avere **al massimo una default export**. Tecnicamente, potremmo avere sia il default che il named export nello stesso modulo, ma in pratica *si tende a non farlo*.

Un modulo generalmente usa named export **o** default export.

Import con alias

A volte è utile rinominare un import per evitare conflitti di nomi o per renderlo più chiaro:

math.js

```
export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}
```

main.js

```
import { add as sum, subtract as diff } from './math.js';

console.log(sum(5, 3));    // 8
console.log(diff(10, 4));  // 6
```

Possiamo anche importare **tutto** come un unico oggetto namespace:

```
import * as math from './math.js';

console.log(math.add(5, 3));       // 8
console.log(math.subtract(10, 4)); // 6
```

Questo è utile quando un modulo esporta molte funzioni e vogliamo raggrupparle sotto un unico nome.

Caricare moduli nel browser

Abbiamo visto che i moduli richiedono una sintassi specifica (`import`/`export`) che non è compatibile con i vecchi script. Per usare i moduli nel browser, dobbiamo indicare che il nostro script è un modulo; per farlo, usiamo l'attributo `type="module"` nel tag `<script>`.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>My page</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module">
      import { sum } from './math.js';

      alert(sum(5, 3)); // 8
    </script>
  </body>
</html>
```

math.js

```
export function sum(a, b) {
  return a + b;
}

export function sub(a, b) {
  return a - b;
}

export function mul(a, b) {
  return a * b;
}

export function div(a, b) {
  return a / b;
}
```

Il browser recupera ed elabora automaticamente il modulo importato (e i suoi import se necessario), e infine esegue lo script.

Dipendenze esterne

Quando il nostro codice ha bisogno di funzionalità già scritte da altri, la **riusiamo** invece di riscriverla da zero.

Una **dipendenza esterna** è una **libreria JavaScript** che installiamo nel progetto e che il nostro codice importa e utilizza.

Esempi:

- Una libreria per manipolare date (`date-fns`)
- Una libreria per gestire lo stato dell'applicazione (`Redux`)
- Un framework (`React`, `Vue`)
- Una libreria per convalidare dati (`Yup`)

Come importarle: CDN

Una **CDN** (Content Delivery Network) è un server pubblico che ospita risorse web statiche di vario tipo.

jsDelivr e cdnjs sono due popolari esempi di CDN che ospitano migliaia di librerie JavaScript.

Possiamo importare una libreria direttamente da CDN riportando l'URL nel tag `<script>`:

```
<!-- Includi in HTML -->
<script src="https://cdn.jsdelivr.net/npm/libreria@versione"></script>
```

Oppure con ES Modules:

```
// Import diretto da URL
import libreria from "https://cdn.jsdelivr.net/npm/libreria@versione";
```

Sono molto pratiche per prototipi veloci o per progetti senza build step.

Tuttavia, per progetti più complessi e scalabili, è consigliabile usare un gestore di pacchetti come NPM, che vedremo tra poco.

Node.js: JavaScript fuori dal browser

Fino ad ora abbiamo eseguito JavaScript solo nel **browser**.

Node.js, invece, è un runtime che esegue JavaScript fuori dal browser:

- Puoi creare server web, API, script di automazione
- **NPM** (Node Package Manager) viene installato automaticamente con Node.js
- È il prerequisito per usare importanti strumenti di supporto (Vite, Webpack, etc.)

Possiamo scaricare Node.js da nodejs.org.

Una volta installato, possiamo verificare le versioni di Node.js e NPM:

```
node --version
# v20.11.0 (o simile)

npm --version
# 10.2.4 (o simile)
```

NPM: il gestore di pacchetti per JavaScript

NPM (***Node Package Manager***) è il principale gestore di pacchetti per JavaScript.

- Scarica librerie da un **registry centralizzato** (npmjs.com)
- Gestisce **versioni e dipendenze** automaticamente
- Richiede **Node.js** installato
- Crea file `package.json` e cartella `node_modules/`

NPM in dettaglio: package.json

Quando usi NPM, tutto parte da **package.json**: il “manifesto” del progetto.

Crea un progetto nuovo:

```
npm init -y
# Crea package.json con valori di default
```

Struttura tipica:

```
{
  "name": "ricettario-app",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build"
  },
  "dependencies": {
    "convert": "^5.0.0"
  },
  "devDependencies": {
    "vite": "^5.0.0"
  }
}
```

dependencies vs devDependencies

dependencies: librerie usate in **produzione** (finiscono nel bundle finale)

```
npm install convert
# Aggiunge convert a dependencies
```

Esempio: `convert`, `axios`, `date-fns`, React, Vue, etc.

devDependencies: strumenti usati solo in **sviluppo** (non finiscono nel bundle)

```
npm install --save-dev vite
# Aggiunge vite a devDependencies
```

Esempio: `vite`, `webpack`, `eslint`, `prettier`, etc.

Perché la distinzione? Quando fai il deploy, installi solo `dependencies` (bundle più leggero).

Installare dipendenze

Se cloni un progetto con `package.json`, installa tutto con:

```
npm install
# Legge package.json e installa tutte le dipendenze in node_modules/
```

Aggiungere una nuova dipendenza:

```
npm install convert
# Scarica convert, lo mette in node_modules/, aggiorna package.json
```

Ora puoi usarla nel codice:

```
import convert from 'convert';

const oz = convert(200, 'g').to('oz');
```

Nota: Non serve URL! NPM risolve automaticamente il modulo da `node_modules/`.

Versioning in NPM

Quando installi una libreria, NPM aggiunge la versione in `package.json`:

```
"dependencies": {
  "convert": "^5.0.0"
}
```

Significato dei simboli:

- `^5.0.0` → accetta versioni `>=5.0.0` e `<6.0.0` (aggiornamenti minori)
- `~5.0.0` → accetta versioni `>=5.0.0` e `<5.1.0` (solo patch)
- `5.0.0` → versione esatta (bloccata)

In pratica, `^` è lo standard: ricevi aggiornamenti compatibili automaticamente.

Bundler: perché?

Problema: 50 moduli nel progetto = 50 HTTP requests.

```
index.js
├── api/recipeService.js
│   ├── utils/validators.js
│   └── api/helpers.js
├── components/RecipeList.js
│   └── utils/dateFormat.js
└── 47 altri moduli
```

Ogni richiesta = ritardo. Lento in produzione.

Soluzione: Un **bundler** combina i 50 moduli in **1 (o pochi) bundle**.

- Rimuove codice inutilizzato (**tree-shaking**)
- Minifica (accorcia nomi, rimuove spazi)
- Compresso: **1 richiesta HTTP**, ~50KB (vs 200KB sparsi)

Vite

Vite è un bundler moderno, veloce, zero-config.

Caratteristiche:

- **Zero-config**: funziona di default
- **HMR**: reload automatico durante sviluppo
- **ES Modules nativi**: capisce import/export
- **Ottimizzazione**: tree-shaking, minify automatico

Flusso:

```
Sviluppo:    src/ → npm run dev     → localhost:5173 (HMR veloce)
Produzione:  src/ → npm run build   → dist/ (ottimizzato)
```

Usare Vite

Setup:

```
# Crea progetto Vite
npm create vite@latest my-app -- --template vanilla

# Entra nella cartella
cd my-app

# Installa dipendenze
npm install

# Avvia dev server (localhost:5173)
npm run dev

# Build per produzione (crea dist/)
npm run build
```

Vite è zero-config: non serve `.config.js` per progetti semplici.