Express.js Cheatsheet

Indice Rapido

Introduzione a Express.js

Express.js è un framework web **minimo e flessibile** per Node.js, che fornisce un set di funzionalità robuste per applicazioni web e mobile. È progettato per costruire **single-page, multi-page, e applicazioni web ibride.**

Setup di Base

1. **Inizializza un progetto Node.js (se non lo hai già fatto):**

```
npm init -y
```

2. **Installa Express.js:**

```
npm install express
```

3. **Crea un file server.js (o index.js) e imposta il server:**

```
// server.js
const express = require("express");
const app = express();
```

```
const port = 3000;

app.get("/", (req, res) => {
  res.send("Ciao Mondo da Express!");
});

app.listen(port, () => {
  console.log(`Server in ascolto sulla porta ${port}`);
});
```

4. **Avvia il server:**

   ```
   node server.js
   ```

   Oppure, se hai installato `nodemon` (consigliato per lo sviluppo):

   ```
   nodemon server.js
   ```

Routing

Express.js usa il routing per determinare come un'applicazione risponde a una richiesta del client verso un particolare endpoint, ovvero un URI (o path) e uno specifico metodo di richiesta HTTP (GET, POST, ecc.).

Routing di Base

```
const express = require("express");
const app = express();

// Route GET per la homepage
app.get("/", (req, res) => {
  res.send("Homepage GET");
});

// Route POST per la homepage
app.post("/", (req, res) => {
  res.send("Homepage POST");
});

// Route GET per /users
app.get("/users", (req, res) => {
  res.send("Lista utenti");
});

// Altri metodi HTTP:  app.put(), app.delete(), app.patch(), app.options(),
app.head()
```

Parametri nelle Route

I parametri route sono segmenti di URL nominati che vengono utilizzati per acquisire i valori specificati nella posizione dell'URL.

```
// Route con parametro :userId
app.get("/users/:userId", (req, res) => {
  res.send(`Dettagli utente con ID: ${req.params.userId}`);
});

// Route con più parametri
app.get("/products/:productId/reviews/:reviewId", (req, res) => {
  res.send(`Recensione ${req.params.reviewId} del prodotto
${req.params.productId}`);
});
```

Route con Espressioni Regolari

Le route path possono essere stringhe, pattern di stringhe o espressioni regolari.

```
// Route che matcha path che iniziano con 'abc' e sono seguite da qualcosa
app.get("/ab*cd", (req, res) => {
  res.send("Route che matcha /ab*cd");
});

// Route che matcha 'abcd' o 'abbcd' o 'abbbcd' ecc.
app.get("/ab+cd", (req, res) => {
  res.send("Route che matcha /ab+cd");
});

// Route che matcha 'abcd' o 'abd'
app.get("/ab?cd", (req, res) => {
  res.send("Route che matcha /ab?cd");
});

// Route che matcha 'abcd' o 'abcde' o 'abcdefg' e così via
app.get("/abc(de)?", (req, res) => {
  res.send("Route che matcha /abc(de)?");
});
```

Router Modulari

Per organizzare le route in modo modulare, puoi usare `express.Router()`.

1. **Crea un file separato per le route (es. `users.js`):**

```
// users.js
const express = require("express");
const router = express.Router();

// Route per /users/
router.get("/", (req, res) => {
  res.send("Lista utenti dal router users");
```

```
});

// Route per /users/:userId
router.get("/:userId", (req, res) => {
  res.send(`Dettagli utente ${req.params.userId} dal router users`);
});

module.exports = router;
```

2. **Usa il router nel tuo server principale (`server.js`):**

```
// server.js
const express = require("express");
const app = express();
const usersRouter = require("./users"); // Importa il router users

app.use("/users", usersRouter); // Monta il router users sotto /users

app.get("/", (req, res) => {
  res.send("Homepage");
});

app.listen(3000, () => console.log("Server avviato"));
```

Oggetti Request (req) e Response (res)

Questi oggetti sono **fondamentali** in Express.js.

Oggetto Request (req)

L'oggetto `req` rappresenta la **richiesta HTTP** e ha proprietà per:

- `req.params` : Parametri route (`/users/:userId`)
- `req.query` : Parametri query string (`/users?sort=name`)
- `req.body` : Corpo della richiesta (per POST, PUT, PATCH) - *Richiede middleware come* `body-parser` *o* `express.json()`
- `req.headers` : Header della richiesta
- `req.cookies` : Cookie inviati dal client - *Richiede middleware come* `cookie-parser`
- `req.ip` : Indirizzo IP del client
- `req.path` : Path richiesto
- `req.method` : Metodo HTTP (GET, POST, ecc.)
- `req.url` : URL completo richiesto

Esempio di accesso a `req.params` , `req.query` , `req.body` :

```
app.use(express.json()); // Middleware per parsare il body JSON

app.get("/search", (req, res) => {
  console.log("Query parameters:", req.query); // Es: { keyword: 'express',
category: 'framework' }
  res.send(`Risultati ricerca per: ${req.query.keyword}`);
```

```
});

app.get("/users/:userId", (req, res) => {
  console.log("Route parameters:", req.params); // Es: { userId: '123' }
  res.send(`Dettagli utente ID: ${req.params.userId}`);
});

app.post("/profile", (req, res) => {
  console.log("Request body:", req.body); // Es: { name: 'John', email:
'john@example.com' }
  res.send(`Profilo aggiornato per: ${req.body.name}`);
});
```

Oggetto Response (res)

L'oggetto `res` è usato per **inviare la risposta HTTP** dal server. Metodi comuni:

- `res.send([body])` : Invia una risposta di vario tipo (stringa, buffer, oggetto, array).
- `res.json(obj)` : Invia una risposta JSON.
- `res.render(view [, locals] [, callback])` : Renderizza una view template.
- `res.redirect([status,] path)` : Redireziona a un altro URL.
- `res.status(code)` : Imposta il codice di stato HTTP.
- `res.sendStatus(code)` : Imposta il codice di stato HTTP e invia il testo di stato corrispondente.
- `res.sendFile(path [, options] [, callback])` : Invia un file.
- `res.download(path [, filename] [, options] [, callback])` : Invia un file per il download.
- `res.cookie(name, value [, options])` : Imposta un cookie.
- `res.clearCookie(name [, options])` : Cancella un cookie.
- `res.setHeader(name, value)` : Imposta un header di risposta.
- `res.end([data] [, encoding] [, callback])` : Termina la risposta.

Esempi di utilizzo di `res` :

```
app.get("/data", (req, res) => {
  const data = { message: "Dati dal server", status: "success" };
  res.json(data); // Invia JSON
});

app.get("/file", (req, res) => {
  res.sendFile(__dirname + "/public/index.html"); // Invia un file HTML
});

app.get("/redirect-me", (req, res) => {
  res.redirect("/destination"); // Redireziona
});

app.get("/custom-status", (req, res) => {
  res.status(404).send("Pagina non trovata"); // Imposta status e invia messaggio
});

app.get("/download-file", (req, res) => {
  res.download(__dirname + "/public/download.txt", "my-file.txt"); // Invia file per
```

```
download
});
```

Middleware

I middleware sono **funzioni** che hanno accesso all'oggetto `req` (richiesta), `res` (risposta) e alla funzione `next()` nel ciclo richiesta-risposta dell'applicazione. Possono:

- Eseguire qualsiasi codice.
- Apportare modifiche agli oggetti request e response.
- Terminare il ciclo request-response.
- Chiamare il prossimo middleware nello stack.

Utilizzo di Middleware

Si usa `app.use()` per montare middleware. Può essere applicato a:

- **Tutte le route:** `app.use(middlewareFunction)`
- **Route specifiche:** `app.use('/path', middlewareFunction)`
- **Metodi HTTP specifici:** `app.get('/path', middlewareFunction, routeHandler)` (middleware applicato solo a questa route GET)

Esempio di middleware base:

```
const loggerMiddleware = (req, res, next) => {
  console.log(`Richiesta: ${req.method} ${req.url}`);
  next(); // Importante: chiama next() per passare al prossimo middleware o route
handler
};

app.use(loggerMiddleware); // Applica loggerMiddleware a tutte le route
```

Middleware Built-in

Express.js include alcuni middleware built-in:

- `express.static(root, [options])` : Per servire file statici (HTML, CSS, immagini, etc.).
- `express.json([options])` : Per parsare il corpo della richiesta in formato JSON.
- `express.urlencoded({ extended: true })` : Per parsare il corpo della richiesta in formato URL-encoded (form).

Esempio di `express.static` :

```
app.use(express.static("public")); // Serve i file dalla cartella 'public' (es:
public/index.html accessibile come /index.html)
```

Esempio di `express.json` e `express.urlencoded` :

```
app.use(express.json()); // Parsa application/json request bodies
app.use(express.urlencoded({ extended: true })); // Parsa application/x-www-form-
urlencoded request bodies
```

```
app.post("/submit", (req, res) => {
  console.log("Body:", req.body); // Dati parsati da JSON o URL-encoded
  res.send("Dati ricevuti!");
});
```

Body Parser Middleware

Anche se `express.json()` e `express.urlencoded()` sono ora inclusi in Express, prima si usava spesso `body-parser`. Se vedi codice vecchio, potresti incontrarlo.

```
const bodyParser = require("body-parser");

app.use(bodyParser.json()); // Parsa application/json
app.use(bodyParser.urlencoded({ extended: true })); // Parsa application/x-www-form-urlencoded
```

Middleware Statici

`express.static()` è un middleware statico molto utile per servire file come immagini, CSS, JavaScript dal tuo server.

```
// Serve file statici dalla cartella 'public' accessibile tramite URL /static
app.use("/static", express.static("public"));
// Ora i file in 'public' sono accessibili come /static/nomefile.estensione
```

Middleware Personalizzati

Puoi creare i tuoi middleware per loggare, autenticare, gestire errori, etc.

Esempio di middleware di autenticazione:

```
const authenticate = (req, res, next) => {
  const apiKey = req.headers["x-api-key"];
  if (apiKey === "miaChiaveSegreta") {
    next(); // Utente autenticato, prosegui
  } else {
    res.status(401).send("Autenticazione fallita");
  }
};

app.get("/protected", authenticate, (req, res) => {
  res.send("Risorsa protetta, accesso consentito!");
});
```

Gestione degli Errori

Puoi definire middleware per la gestione degli errori. Questi middleware hanno **quattro parametri**: `(err, req, res, next)`. Express li riconosce come middleware di gestione degli errori.

```
app.get("/error", (req, res, next) => {
  throw new Error("Qualcosa è andato storto!"); // Forza un errore
});

// Middleware di gestione degli errori (deve essere definito DOPO le route)
app.use((err, req, res, next) => {
  console.error(err.stack); // Log dell'errore (utile in sviluppo)
  res.status(500).send("Qualcosa si è rotto!"); // Risposta di errore generica per
l'utente
});
```

CORS (Cross-Origin Resource Sharing)

CORS è un meccanismo di sicurezza del browser che blocca le richieste HTTP cross-origin (richieste da un dominio diverso da quello in cui è servita la pagina web). Se il tuo frontend è su un dominio diverso dal tuo backend Express.js, avrai bisogno di CORS.

1. **Installa il pacchetto `cors`:**

   ```
   npm install cors
   ```

2. **Usa il middleware `cors` nella tua app Express:**

   ```
   const cors = require("cors");
   app.use(cors()); // Abilita CORS per tutte le origini e route (in sviluppo,
   in produzione configura in modo più restrittivo)

   // Esempio più specifico (solo per una origine):
   // app.use(cors({
   //   origin: 'http://mio-frontend.com'
   // }));
   ```

Template Engine (Esempio con Handlebars)

I template engine ti permettono di generare HTML dinamico lato server. Handlebars è un template engine popolare.

1. **Installa `express-handlebars`:**

   ```
   npm install express-handlebars
   ```

2. **Configura Handlebars come view engine in Express:**

   ```
   const { engine } = require("express-handlebars");

   app.engine("handlebars", engine());
   app.set("view engine", "handlebars");
   ```

```
app.set("views", "./views"); // Cartella dove cercherà i file .handlebars
(default è 'views')
```

3. **Crea una cartella `views` nella root del progetto e un file template Handlebars (es. `home.handlebars`):**

```
<!-- views/home.handlebars -->
<h1>Ciao {{nome}}!</h1>
<p>Benvenuto nel mio sito.</p>
```

4. **Renderizza il template nella route:**

```
app.get("/home", (req, res) => {
  res.render("home", { nome: "Utente Express" }); // Renderizza
'home.handlebars' e passa i dati { nome: ... }
});
```

Autenticazione (Brevemente Passport.js e JWT)

Express.js in sé non gestisce l'autenticazione. Si usano middleware e librerie esterne.

- **Passport.js:** Libreria flessibile per l'autenticazione. Supporta molte strategie (username/password, OAuth, social login, etc.). Richiede configurazione per ogni strategia.
- **JWT (JSON Web Tokens):** Usato per autenticazione stateless. Il server genera un token dopo il login, e il client lo invia nelle richieste successive per autenticarsi. Pacchetti come `jsonwebtoken` (per generare e verificare token) e `express-jwt` (middleware per proteggere le route con JWT).

Esempio concettuale JWT:

1. **Login:** Utente invia credenziali, server verifica, genera JWT e lo invia al client.
2. **Richieste successive:** Client invia JWT nell'header `Authorization`, middleware `express-jwt` verifica il token, se valido la route protetta viene eseguita.

NPM e Gestione dei Pacchetti

Comandi NPM utili in Express.js e in generale per Node.js:

- `npm init -y` : Inizializza un nuovo progetto Node.js (crea `package.json`).
- `npm install <nome-pacchetto> [--save | --save-dev]` : Installa un pacchetto. `--save` (o `-S`) per dipendenze di produzione, `--save-dev` (o `-D`) per dipendenze di sviluppo.
- `npm uninstall <nome-pacchetto>` : Disinstalla un pacchetto.
- `npm list` : Elenca i pacchetti installati.
- `npm update <nome-pacchetto>` : Aggiorna un pacchetto alla versione più recente.
- `npm install` : Installa tutte le dipendenze elencate in `package.json` (dopo aver clonato un progetto, ad esempio).

nodemon

`nodemon` è un utility che **monitora** le modifiche nei file del tuo progetto Node.js e **riavvia automaticamente** il server. Molto utile in fase di sviluppo per non dover riavviare manualmente il server ad

ogni modifica.

1. **Installa `nodemon` globalmente (una sola volta):**

   ```
   npm install -g nodemon
   ```

2. **Avvia il server con `nodemon` invece di `node`:**

   ```
   nodemon server.js
   ```