



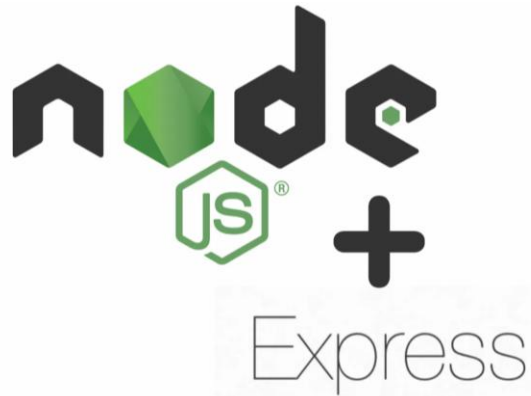
# UNIVERSITÀ DEGLI STUDI DI CAGLIARI

CORSO DI LAUREA IN INFORMATICA

Silvia Maria Massa – [silviam.massa@unica.it](mailto:silviam.massa@unica.it)

Fondamenti di Programmazione Web

Server



# Node.js – perchè lo usiamo?

- **Condivisione del linguaggio tra Frontend e Backend**
  - semplifica la gestione del codice, la manutenzione e favorisce la coerenza nello sviluppo dell'applicazione
- **Vasto assortimento di librerie e moduli**
  - accelera lo sviluppo e fornisce soluzioni preconfezionate per molte esigenze
- **Gestione asincrona delle richieste**
  - gestisce un gran numero di connessioni simultanee in modo efficiente (chat in tempo reale, IoT, streaming di dati)
- **Community attiva e Supporto**
- **Utilizzato da diverse aziende tra cui**



# Node.js - Introduzione

- Non è un linguaggio o un framework
- **JavaScript runtime environment:** fornisce un ambiente per l'esecuzione di codice JavaScript al di fuori di un browser web
- **Open-source:** chiunque può visualizzare, utilizzare, modificare e distribuire il codice sorgente secondo i termini della licenza open-source
- **Multipiattaforma:** può essere eseguito su diversi sistemi operativi, come Windows, macOS e vari tipi di Unix/Linux
- È costruito sul **V8 JavaScript engine** di Google Chrome

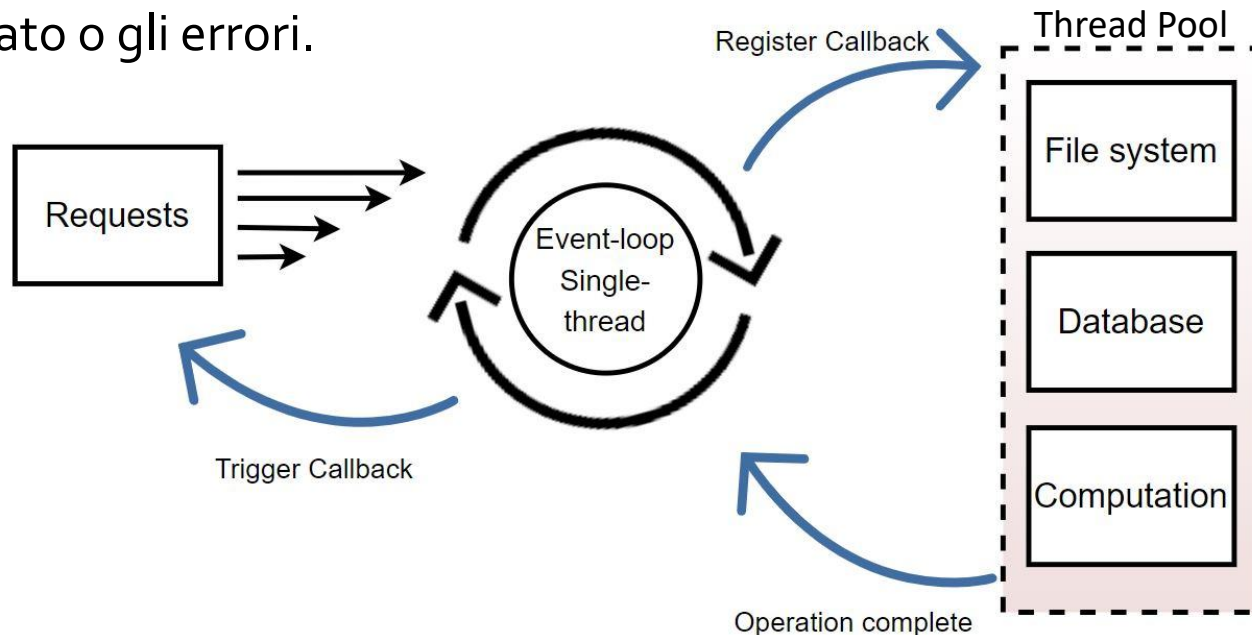


# Node.js – Event-driven, Non-blocking I/O

- È utilizzato per la programmazione lato server (ma non solo) e viene impiegato principalmente per la creazione di **non-blocking, event-driven server** che permettono di gestire in modo efficiente molte connessioni simultanee senza consumare eccessive risorse
- **Event-driven**
  - Il flusso di esecuzione del codice non è tradizionale ma è guidato dagli eventi
  - Un esempio di evento sono le richieste HTTP.
  - Questi eventi sono registrati con funzioni di callback associate e quando si verifica un evento la callback corrispondente viene eseguita in modo asincrono.

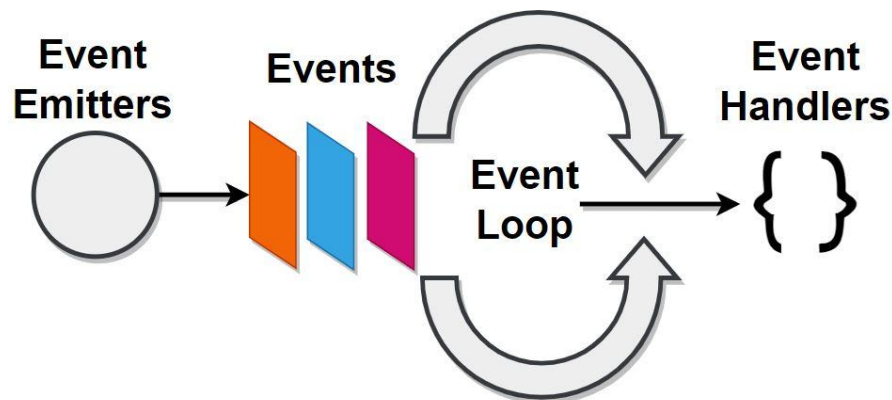
# Node.js – Event-driven, Non-blocking I/O

- **Non-blocking I/O**
  - Quando un'operazione di I/O bloccante viene avviata, come la lettura/scrittura da database, Node.js non si blocca per aspettare il completamento di questa operazione di I/O ma passa a eseguire altre operazioni.
  - Quando l'operazione di I/O eseguita in background è completata, viene chiamata la funzione di callback associata a quell'operazione per gestire il risultato o gli errori.



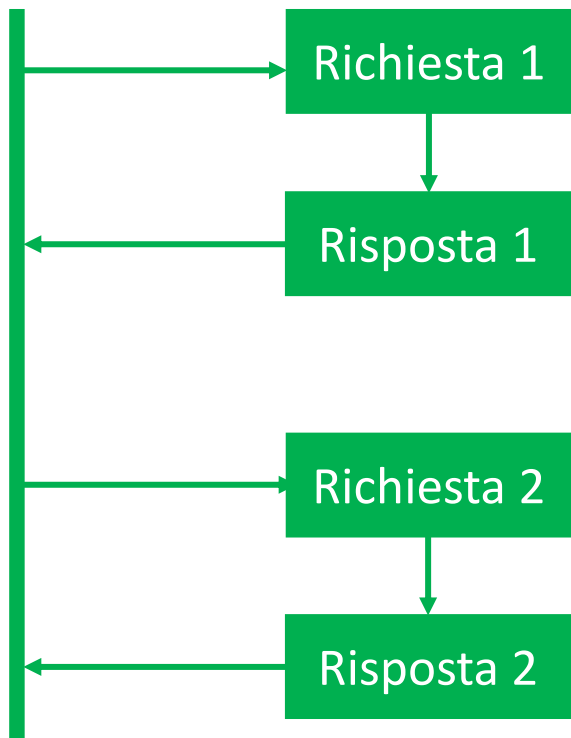
# Node.js – Architettura, Event loop

- L'**event loop** è il cuore dell'architettura event-driven e controlla continuamente gli eventi in attesa in un ciclo.
1. Quando viene registrato un nuovo evento, questo viene aggiunto alla coda degli eventi.
  2. L'event loop preleva questi eventi dalla coda ed esegue singolarmente le callback corrispondenti.
  3. Se una callback richiede tempo per essere completata, non blocca l'intero programma, ma nel frattempo possono essere elaborati altri eventi.



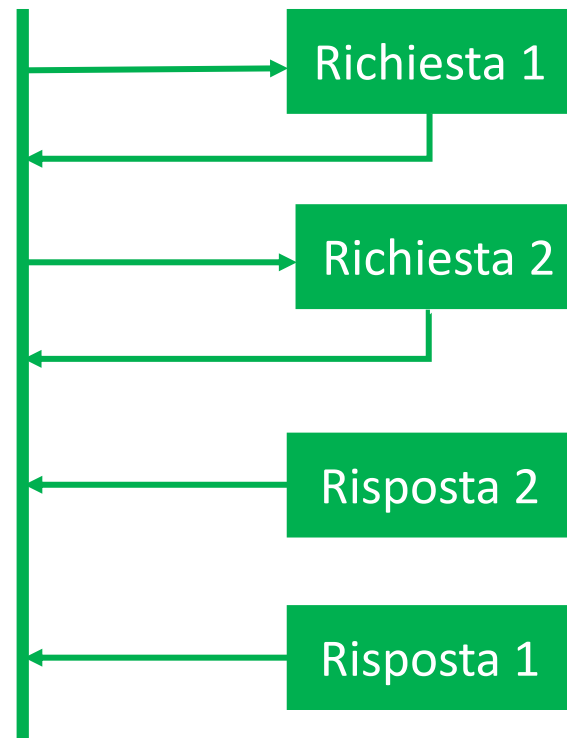
# Node.js – Programmazione asincrona

## Approccio sincrono



VS

## Approccio asincrono



- L'esecuzione avviene riga per riga.
- Le operazioni di lunga durata bloccano l'esecuzione del codice.

- Il codice viene eseguito al termine di un'attività in background.
- L'esecuzione non attende che un task asincrono finisca il suo lavoro.

# Node.js – Programmazione asincrona

Approccio sincrono

```
const fs = require('fs'); //importiamo il modulo per interagire con il file system  
const input = fs.readFileSync ('input.txt', 'utf-8');  
console.log(input);
```

- l'esecuzione aspetta la ricezione dei dati prima di effettuare la stampa su console

Approccio asincrono (utilizzando le callback)

```
const fs = require('fs');  
fs.readFile('input.txt', 'utf-8', (err,data) => {  
    if (err) { console.error('Errore nella lettura del file:',err);  
    else { console.log(data); }  
});  
console.log('Reading file----');
```

- l'azione da effettuare una volta ottenute le informazioni richieste non è scritta dopo la prima, ma è passata come parametro alla funzione readFile sotto forma di funzione callback

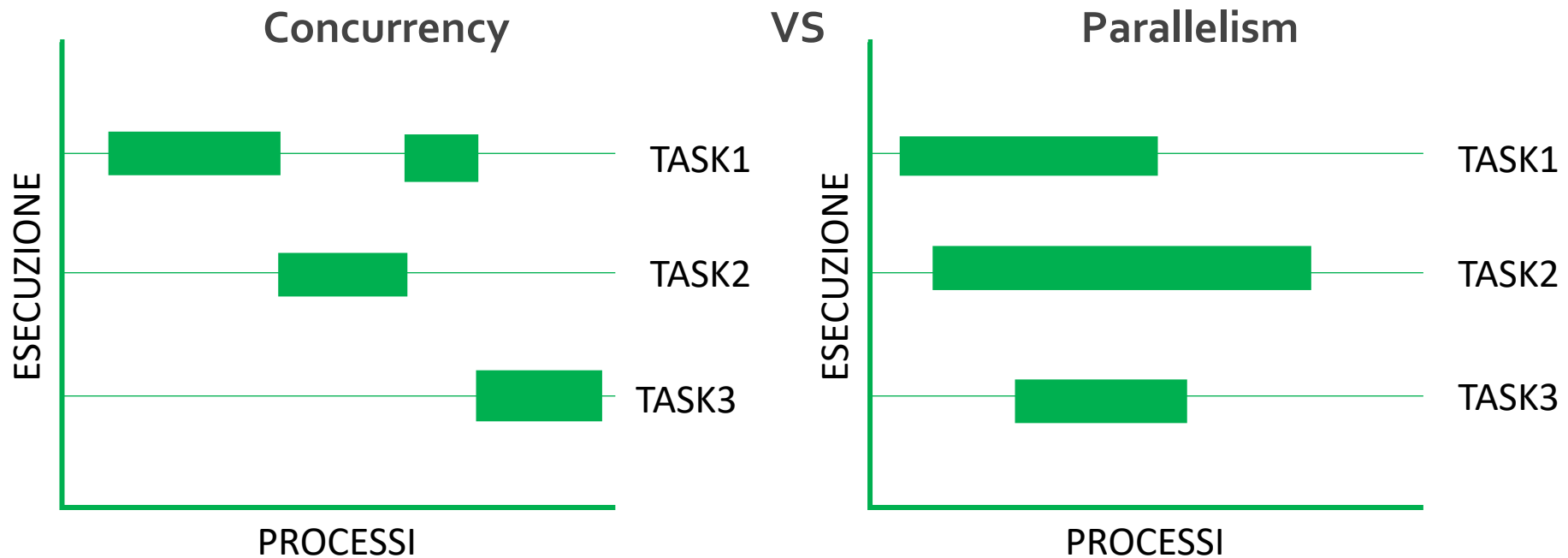


# Node.js – Architettura, Single-threaded

- Multi-threaded processing model
  - Viene scelto un thread ogni volta che viene fatta una richiesta fino a quando tutti i thread sono esauriti. Quando ciò accade il server deve attendere che un thread occupato si liberi di nuovo.
  - Il fatto che il server abbia un pool di thread limitato e che l'elaborazione è sincrona e sequenziale può rendere le applicazioni lente e inefficienti.
  - Può diventare un problema soprattutto se l'applicazione deve gestire un numero elevato di richieste concorrenti da parte dei client.
- **Node.js utilizza un single-threaded processing model**
  - Elabora ogni richiesta usando un singolo thread principale, utilizzando l'event loop per eseguire operazioni di input/output bloccanti in modo non bloccante utilizzando sei thread secondari.

# Node.js – Architettura, Concurrency

- Poiché non si utilizza un multi-threaded processing model, si esclude il parallelismo
- Utilizzando un single-threaded processing model invece si possono eseguire più codici alla volta utilizzando la **modalità concorrente**



# Node.js – Programmazione asincrona

- L'utilizzo esclusivo di callback per gestire il codice asincrono può portare rapidamente a un codice difficile da leggere e ingestibile.
- Esempio: Il secondo file letto dipende dal primo letto, la scrittura nel file dipende dalla lettura del secondo e terzo file.

```
const fs = require('fs');
fs.readFile('start.txt', 'utf-8', (err, data1) => {
  if (err) { console.error('Error reading start.txt:', err); }
  else { fs.readFile(`${data1}.txt`, 'utf-8', (err, data2) => {
    if (err) { console.error('Error reading ' + data1 + '.txt:', err); }
    else { fs.readFile('append.txt', 'utf-8', (err, data3) => {
      if (err) { console.error('Error reading append.txt:', err); }
      else { fs.writeFile('append.txt', `${data2} ${data3}`, 'utf-8', (err) => {
        if (err) { console.error('Error writing to append.txt:', err); }
        else { console.log('Your file has been saved'); }
      }); }
    }); }
  }); }
});
```

Codice da non replicare

**CALLBACK  
HELL**

Possiamo usare strumenti più avanzati per gestire il codice asincrono come le **promise**.

# Promise



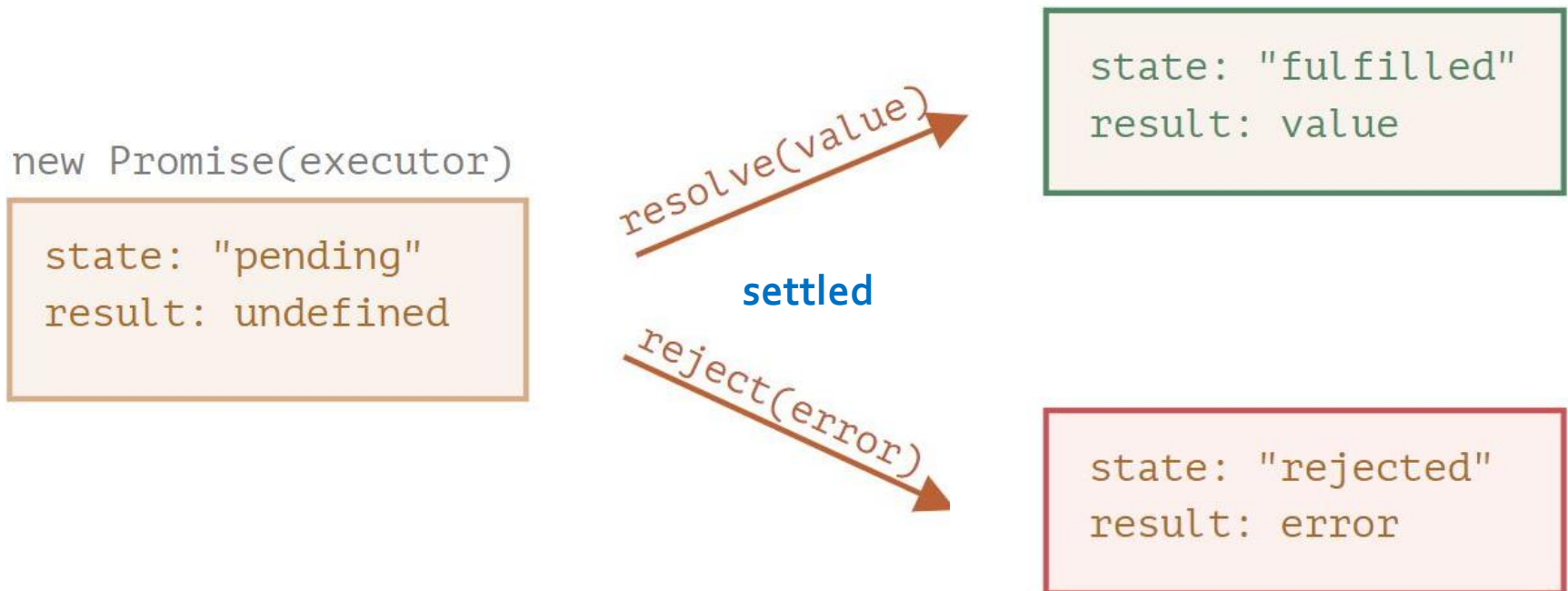
- Una **promise** è uno speciale oggetto JavaScript che collega:
  - Un **producing code** fa qualcosa e richiede tempo.
  - Un **consuming code** vuole il risultato del producing code una volta che è pronto.
- La sintassi del costruttore per un oggetto promise è:

```
let promise = new Promise(function(resolve, reject) {  
    // producing code  
});
```

  - La funzione passata a `new Promise` è chiamata **executor**. Quando la promise è creata, questa funzione viene eseguita automaticamente.
  - Gli argomenti della funzione executor, **resolve** e **reject**, sono delle callback fornite da JavaScript stesso.

# Promise

- L'oggetto promise restituito ha le proprietà interne **state** e **result** che variano quando viene invocato `resolve( value )` o `reject( error )`.
- Caso 1: **resolve(value)** viene chiamata se il processo termina correttamente, col risultato `value`.
- Caso 2: **reject(error)** viene chiamata se si verifica un errore, `error` è l'oggetto



# Promise - .then .catch

- Il consuming code utilizzerà il risultato del producing code una volta pronto attraverso **.then**  
    promise.**then**(  
        //consuming code  
    );
- Per gestire l'errore utilizziamo **.catch**  
    promise.**catch**(  
        //errorHandlingFunction  
    );



# Promise - .then .catch

```
const fs = require('fs').promises;
fs.readFile('start.txt', 'utf-8')
  .then(data1 => {
    return fs.readFile(`${data1}.txt`, 'utf-8');
  })
  .then(data2 => {
    return fs.readFile('append.txt', 'utf-8')
      .then(data3 => {
        return fs.writeFile('append.txt', `${data2} ${data3}`, 'utf-8');
      });
  })
  .then(() => {
    console.log('Your file has been saved');
  })
  .catch(err => {
    console.error('Error:', err);
  });
```

L'API `fs.promises` fornisce un insieme alternativo di metodi asincroni per il file system che restituiscono oggetti Promise anziché utilizzare callback. L'API è accessibile tramite `require('fs').promises`.

CALLBACK

```
const fs = require('fs');
fs.readFile('start.txt', 'utf-8', (err, data1) => {
  if (err) { console.error('Error reading start.txt:', err); }
  else { fs.readFile(`${data1}.txt`, 'utf-8', (err, data2) => {
    if (err) { console.error('Error reading ' + data1 + '.txt:', err); }
    else { fs.readFile('append.txt', 'utf-8', (err, data3) => {
      if (err) { console.error('Error reading append.txt:', err); }
      else { fs.writeFile('append.txt', `${data2} ${data3}`, 'utf-8', (err) => {
        if (err) { console.error('Error writing to append.txt:', err); }
        else { console.log('Your file has been saved'); }
      }); }
    }); }
  }); }
});
```

# Async/await

- Async e await permettono di scrivere codice asincrono più facile da leggere e da scrivere.
- **async** viene messa prima delle funzioni e ha due effetti:
  - Ritorna una Promise.
  - Permette di usare await al suo interno.
- La parola chiave **await** prima di una Promise fa attendere JavaScript fino a quando la Promise diventa settled:
  - Se c'è un errore viene generata l'eccezione
  - Altrimenti, ritorna il risultato.
- Quando dobbiamo attendere per più Promise contemporaneamente utilizziamo **Promise.all**.
- Se una qualsiasi delle promise è respinta (rejected), Promise.all viene immediatamente respinta (rejects).  
**let promise = Promise.all([...promises...]);**



# Async/await

- Esempio singola Promise :

```
const fs = require('fs').promises;
```

```
async function processFiles() {  
  try {
```


```
    const data1 = await fs.readFile('start.txt', 'utf-8');  
    const data2 = await fs.readFile(`${data1}.txt`, 'utf-8');  
    const data3 = await fs.readFile('append.txt', 'utf-8');  
    await fs.writeFile('append.txt', `${data2} ${data3}`, 'utf-8');  
    console.log('Your file has been saved');
```

```
  }  
  catch (err) {  
    console.error('Error:', err);  
  }  
}
```

```
processFiles();
```

```
const fs = require('fs').promises;  
fs.readFile('start.txt', 'utf-8')  
  .then(data1 => {  
    return fs.readFile(`${data1}.txt`, 'utf-8');  
  })  
  .then(data2 => {  
    return fs.readFile('append.txt', 'utf-8')  
      .then(data3 => {  
        return fs.writeFile('append.txt', `${data2} ${data3}`, 'utf-8');  
      });  
  })  
  .then(() => {  
    console.log('Your file has been saved');  
  })  
  .catch(err => {  
    console.error('Error:', err);  
  });
```

PROMISES - .then(.catch





# Async/await

- Esempio Promise.all :

```
const fs = require('fs').promises;
```

```
async function processFiles() {  
  try {  
    const data1 = await fs.readFile('start.txt', 'utf-8');  
    const [data2, data3] = await Promise.all(  
      [  
        fs.readFile(`${data1}.txt`, 'utf-8'),  
        fs.readFile('append.txt', 'utf-8')  
      ]  
    );  
    await fs.writeFile('append.txt', `${data2} ${data3}`, 'utf-8');  
    console.log('Your file has been saved');  
  }  
  catch (err) {  
    console.error('Error:', err); }  
}  
  
processFiles();
```



# Semplificare il lavoro con Node.js grazie al framework Express

# Node.js – NPM

- Una volta installato Node.js avremo a disposizione **npm** (Node Package Manager)
- Ogni progetto ha un file **package.json**. Se vogliamo installare tutte le dipendenze (cioè tutti i moduli utili all'applicazione) eseguiamo

**npm install**

tutti i moduli necessari saranno installati nella cartella **node\_modules** creandola se non esistente già.

- Possiamo installare un pacchetto specifico eseguendo

**npm install <name\_package>**

(viene aggiunto <name\_package> nel file **package.json**)

- Possiamo installare una versione specifica di un pacchetto eseguendo

**npm install <name-package>@<version>**

- Se vogliamo aggiornare i pacchetti eseguiamo

**npm update**

**npm update <name\_package>**

# Express.js

- **Express** è un framework che lavora sopra le funzionalità del server web Node.js per semplificare le sue API e aggiungere nuove utili funzionalità.
- Rende più facile organizzare le funzionalità dell'applicazione con Routing e Middleware.
- Installazione di Express per lo sviluppo back-end:
  - 1) **mkdir myapp** – creiamo la directory di lavoro per la nostra applicazione
  - 2) **cd myapp** – ci posizioniamo al suo interno
  - 3) **npm init** - creiamo un file **package.json** per l'applicazione  
di default l'entry point sarà **index.js**
  - 4) **npm install express** – installiamo il framework Express nel nostro progetto Node.js.

# Express.js – Hello world!

- Creiamo l'entry point file index.js e inseriamoci questo codice

```
const express = require('express'); // importiamo il modulo con nome 'express'  
const app = express(); // creiamo un'istanza dell'applicazione Express  
const port = 3000; // definiamo la porta su cui l'applicazione ascolterà le richieste
```

```
//definiamo il gestore per la richiesta GET sulla radice
```

```
app.get('/', (req, res) => {  
    res.send('Hello World!');  
});
```

```
// avviamo il server
```

```
app.listen(port, () => { console.log (`Server is running on port 3000`); });
```

Node.js è il runtime che esegue il codice JavaScript.  
Express.js fornisce il framework per la gestione delle richieste e delle risposte HTTP.

# Express.js – Hello world!

- Con il comando **node index.js** eseguiamo l'applicazione.

**nodemon** è uno strumento che aiuta a sviluppare applicazioni basate su Node.js, riavviando automaticamente l'applicazione node quando vengono rilevate modifiche ai file nella directory.

**npm install nodemon**

Per eseguire l'applicazione usiamo al posto di node seguito dal nome dell'applicazione il comando

**nodemon nome\_applicazione**

- L'applicazione avvia un server e resta in ascolto sulla porta 3000 per le connessioni
- Tramite **http://localhost:3000/** possiamo visualizzare l'output sul browser .
- L'applicazione risponde con "Hello World!" per le richieste all'URL root (/).
- Per qualsiasi altro percorso, risponderà con il messaggio **404 not found**



# Routing



# Express.js - Routing

- **Routing** si riferisce alla determinazione del modo in cui un'applicazione risponde a una richiesta del client (HANDLER) a un particolare endpoint (combinazione di PATH e METHOD).
- La definizione di una **route** ha la seguente struttura:

**app.METHOD(PATH, HANDLER)**

Dove:

- **app** è un'istanza della classe express
  - **METHOD** è un metodo di richiesta HTTP in minuscolo (GET, POST, e così via)
  - **PATH** è un percorso sul server (un URI)
  - **HANDLER** è la funzione eseguita quando si trova una corrispondenza per la route
- Ogni route può avere una o più handler function, le quali vengono eseguite quando si trova una corrispondenza per la route.

# Express.js - Routing

- **Esempi di route semplici**

- Risponde con Hello World! sulla homepage dell'applicazione (/):

```
app.get('/', function (req, res) {  
    res.send('GET request to the homepage');  
});
```

- Risponde alla richiesta POST sulla homepage dell'applicazione (/): :

```
app.post('/', function (req, res) {  
    res.send('POST request to the homepage');  
});
```

# Express.js – req e res

- req e res sono oggetti che rappresentano rispettivamente la richiesta e la risposta HTTP.
- **req** (abbreviazione di **request**) è l'oggetto che rappresenta la richiesta fatta dal client (es. il browser).
- Contiene informazioni come i dati inviati nel corpo della richiesta (**req.body**)
- **res** (abbreviazione di **response**) è l'oggetto usato per inviare la risposta al client.
- Puoi servire per:
  - inviare una risposta testuale o HTML (**res.send()**)
  - inviare una risposta JSON (**res.json()**)
  - impostare lo status HTTP (es. **res.status(404)**)

# Express.js – Routing PATHS

- I percorsi delle route possono essere semplici stringhe ma anche **regular expressions**.
- Una **regular expression** è una **sequenza di simboli** che descrive un **pattern** all'interno di una stringa.
- Esempio percorso basato su **regular expressions** :

Questo percorso corrisponde a `/fly/`, `/butterfly/`, `/dragonfly/`, e così via.

```
app.get(/.*fly$/, (req, res) => {  
    res.send('/.*fly$/');  
});
```

corrisponde a qualsiasi stringa che termina con la sequenza di caratteri "fly"  
.\*corrisponde a zero o più caratteri di qualsiasi tipo.  
\$ questo carattere indica la fine della stringa

# Express.js – Routing PATHS parameters

- Nell'URL della route possono esserci dei **parametri**
- Possono essere identificati velocemente perché nel route PATH hanno la forma **: <nome\_parametro>**
- Nell'oggetto **req.params** viene specificato il valore dei parametri utilizzando la forma **chiave: valore**, dove chiave corrisponde a **<nome\_parametro>**
- Esempio:
  - Route path: `/users/:userId/books/:bookId`
  - Request URL: `http://localhost:3000/users/34/books/8989`
  - `req.params: { "userId": "34", "bookId": "8989" }`
  - ```
app.get('/users/:userId/books/:bookId', (req, res) => {  
    let bookId = req.params.bookId;  
    res.send(`Hai selezionato il libro ${bookId}`);  
});
```

# Express.js – Routing PATHS parameters


- Poiché il **trattino** (-) e il **punto** (.) vengono interpretati letteralmente, possono essere utilizzati insieme ai parametri di percorso per scopi utili.
- Esempi:
  - Route path: `/flights/:from-:to`  
Request URL: `http://localhost:3000/flights/LAX-SFO`  
`req.params: { "from": "LAX", "to": "SFO" }`
  - Route path: `/plantae/:genus.:species`  
Request URL: `http://localhost:3000/plantae/Prunus.persica`  
`req.params: { "genus": "Prunus", "species": "persica" }`

# Express.js – Routing METHOD `.all()`


- Esiste un metodo di routing speciale, `app.all()`.
- Questo metodo viene utilizzato per caricare funzioni **middleware** su un percorso per tutti i metodi di richiesta HTTP.
- Esempio:

Vogliamo che l'HANDLER venga eseguito per tutti i metodi di richiesta HTTP (GET, POST, e così via) alla route `"/secret"`

```
app.all('/secret', function (req, res, next) {  
  console.log('Accessing the secret section ...');  
  next(); // pass control to the next handler  
});
```



```
app.get('/secret', function (req, res) {  
  res.send('GET req to the /secret route');  
});
```



```
app.post('/secret', function (req, res) {  
  res.send('POST req to the /secret route');  
});
```

# Express.js – Routing HANDLERS

- È possibile fornire molteplici funzioni di callback che si comportino come middleware per gestire una richiesta.
- La sola eccezione è rappresentata dal fatto che queste callback potrebbero richiamare **next('route')** per ignorare le callback della route restanti.
- È possibile utilizzare questo meccanismo per **imporre pre-condizioni** su una route, quindi, passare il controllo a route successive, nel caso non ci siano motivi per proseguire con la route corrente.
- Gli **handler di route** possono avere il formato di una funzione di callback, di un array di funzioni di callback o di combinazioni di entrambi.



# Express.js – Routing HANDLERS

## ■ Esempi:

- Gestione di una route attraverso **una singola funzione di callback**

```
app.get('/example/a', function (req, res) {  
    res.send('Hello from A!');  
});
```

- Gestione di una route attraverso **più funzioni di callback** (bisogna assicurarsi di specificare next)

```
app.get('/example/b', function (req, res, next) {  
    console.log('the response will be sent by the next function ...');  
    next();  
}, function (req, res) {  
    res.send('Hello from B!');  
});
```

# Express.js – Routing HANDLERS

- Gestione di una route attraverso **un array di funzioni callback**

```
var cbo = function (req, res, next) {  
    console.log('CB0');  
    next();  
}
```

```
var cb1 = function (req, res, next) {  
    console.log('CB1');  
    next();  
}
```

```
var cb2 = function (req, res) {  
    res.send('Hello from CB2!');  
}
```

```
app.get('/example/c', [cbo, cb1, cb2]);
```

# Express.js – Routing HANDLERS

- Gestione di una route attraverso **una combinazione di funzioni indipendenti e array di funzioni**

```
var cbo = function (req, res, next) {  
    console.log('CB0');  
    next();  
}
```

```
var cb1 = function (req, res, next) {  
    console.log('CB1');  
    next();  
}
```

```
app.get('/example/d', [cbo, cb1], function (req, res, next) {  
    console.log('the response will be sent by the next function ...');  
    next();  
}, function (req, res) {  
    res.send('Hello from D!');  
});
```

# Express.js – Routing METHODS

- Esistono diversi metodi da utilizzare sull'oggetto risposta (res) che possono inviare una risposta al client e terminare il ciclo richiesta-risposta.
- Se nessuno di questi metodi viene richiamato da un handler di route, la richiesta del client verrà lasciata in sospeso!

| METODO                                                                                                                          | DESCRIZIONE                               |
|---------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|
| <b>res.download()</b> es. <code>res.download('./files/example.pdf')</code>                                                      | Richiede il download di un file           |
| <b>res.end()</b>                                                                                                                | Termina il processo di risposta           |
| <b>res.json()</b> es. <code>res.json({<br/>  msg: 'Questo è un esempio di risposta JSON',<br/>  status: 'success'<br/>})</code> | Invia una risposta in formato JSON        |
| <b>res.redirect()</b> es. <code>res.redirect('/login')</code>                                                                   | Reindirizza una richiesta                 |
| <b>res.send()</b> es. <code>res.send('Pagina non trovata')</code>                                                               | Invia una risposta in diversi formati     |
| <b>res.sendFile()</b> es.<br><code>res.sendFile('path/assoluto/example.pdf')</code>                                             | Invia un file                             |
| <b>res.status()</b> es. <code>res.status(404)</code>                                                                            | Imposta il codice di stato della risposta |

# Express.js – Routing `app.route()`

- È possibile creare HANDLER di route concatenabili (route modulari) per un PATH, utilizzando **`app.route()`**.
- Poiché il PATH è specificato in un'unica posizione si riduce la ridondanza del codice e si diminuisce la possibilità di avere errori di battitura.
- Esempio

```
app.route('/book')  
  .get((req, res) => {  
    res.send('Get a random book');  
  })  
  .post((req, res) => {  
    res.send('Add a book');  
  })  
  .put((req, res) => {  
    res.send('Update the book');  
  });
```

# Express.js – Routing express.Router

- Utilizzare la funzione **express.Router** per creare HANDLER di route modulari e montabili.
- **Esempio - creiamo un file birds.js nella directory principale**

```
const express = require('express');  
const router = express.Router();  
const timeLog = (req, res, next) => {  
    console.log('Time: ', Date.now());  
    next();  
};  
router.use(timeLog);  
router.get('/', (req, res) => {  
    res.send('Birds home page');  
});  
router.get('/about', (req, res) => {  
    res.send('About birds');  
});  
module.exports = router;
```

middleware specifico per questo router  
che stampa per ogni richiesta il time

montiamo il middleware in router

creiamo una route per gestire le  
richieste alla homepage

creiamo una route per gestire le  
richieste a /about

trattiamo il router come un modulo

# Express.js – Routing express.Router

- Esempio - montiamo ora il modulo router nell'applicazione

```
const birds = require('./birds');
```

```
// ...
```

```
app.use('/birds', birds);
```

importiamo il modulo router specificandone il percorso

montiamo il modulo router nell'applicazione

- Il percorso ('/birds') specifica l'URL di base in cui il middleware sarà invocato. Se si omette questo parametro, che è opzionale, il middleware verrà invocato per tutte le richieste.
- birds invece è la funzione middleware

- L'applicazione sarà ora in grado di gestire le richieste a /birds e /birds/about, nonché di chiamare la funzione middleware timeLog specifica per il percorso.