UNIVERSITÀ DEGLI STUDI
DI CAGLIARI
CORSO DI LAUREA IN INFORMATICA

Silvia Maria Massa – silviam.massa@unica.it

Fondamenti di Programmazione Web

Server

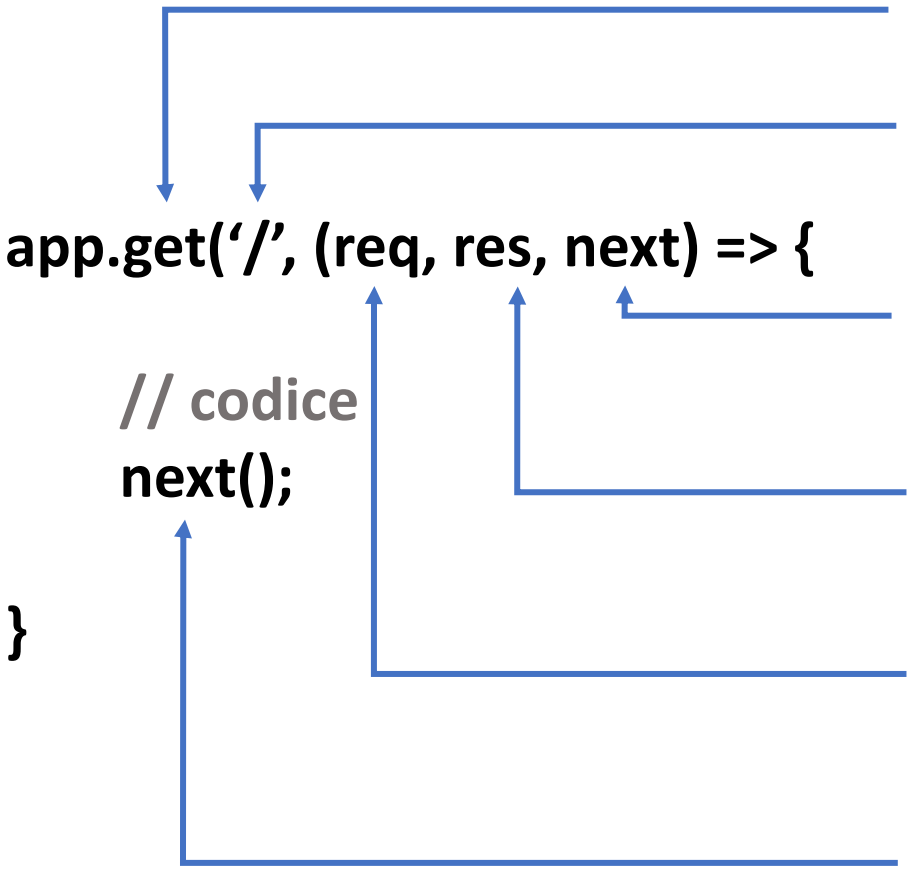
Middleware

Express.js – Middleware

- Le funzioni **middleware** sono funzioni che hanno accesso all'oggetto richiesta (**req**), all'oggetto risposta (**res**), e alla funzione **next** nel ciclo richiesta-risposta dell'applicazione.
- La funzione **next** è una funzione che, quando viene invocata, esegue il middleware successivo a quello corrente.
- Le funzioni middleware possono eseguire i seguenti compiti:
  - Eseguire qualsiasi codice.
  - Apportare modifiche agli oggetti richiesta e risposta.
  - Terminare il ciclo richiesta-risposta.
  - Chiamare il middleware successivo nello stack.
- Se la funzione middleware corrente non termina il ciclo richiesta-risposta, deve chiamare next() per passare il controllo alla funzione middleware successiva. **Altrimenti, la richiesta verrà lasciata in sospeso!**

Express.js – Middleware

- Vediamo gli elementi di una chiamata di funzione middleware con un esempio.



- Metodo HTTP per il quale si applica la funzione middleware.
- Percorso (route) per il quale si applica la funzione middleware.
- Argomento callback della funzione middleware, chiamato per convenzione "next".
- Argomento risposta HTTP della funzione middleware, chiamato per convenzione "res".
- Argomento richiesta HTTP della funzione middleware, chiamato per convenzione "req".
- Funzione per eseguire il middleware successivo (non necessaria se terminiamo il ciclo richiesta-risposta).

Express.js – Middleware

- Vediamo come definire e aggiungere delle funzioni middleware all'applicazione **" Hello world!"** che abbiamo visto all'inizio della scorsa lezione attraverso due esempi.
- Nel primo esempio aggiungiamo la funzione middleware **"myLogger".**
- Questa funzione stampa semplicemente "LOGGED" sul terminale quando una richiesta all'applicazione passa attraverso di essa. La funzione middleware è assegnata a una variabile chiamata myLogger.

```
.....
const app = express()
const myLogger = function (req, res, next) {
        console.log('LOGGED');
        next();
};
app.use(myLogger);
app.get('/', (req, res) => {
......
```

Express.js – Middleware

- L'ordine di caricamento del middleware è importante: le funzioni del middleware che vengono caricate per prime vengono anche eseguite per prime.
- Se myLogger viene montato dopo la route che gestisce la richiesta di tipo get al percorso principale '/', la richiesta non lo raggiunge e l'applicazione non stampa il messaggio "LOGGED", perché l'HANDLER della route (app.get('/', (req, res) => {…}) termina il ciclo richiesta-risposta.
- La funzione middleware myLogger stampa semplicemente un messaggio, quindi passa la richiesta alla funzione middleware successiva nello stack chiamando la funzione next().

Express.js – Middleware

- Nel secondo esempio aggiungiamo la funzione middleware **"requestTime".**
- Questa funzione aggiunge una proprietà chiamata requestTime all'oggetto request e gli assegna la data e l'ora in cui è stata ricevuta la richiesta.

```
.....
const app = express();
const requestTime = function (req, res, next) {
        req.requestTime = Date.now();
        next();
};
app.use(requestTime);
app.get('/', (req, res) => {
        let responseText = 'Hello World!<br>';
        responseText+= `<small>Requested at: ${req.requestTime}</small>`;
        res.send(responseText);
});
......
```

Express.js – Middleware

- La funzione di callback, della route che gestisce la richiesta di tipo get al percorso principale '/', utilizza la proprietà che la funzione middleware aggiunge all'oggetto req.
- Quando si effettua una richiesta di tipo get al percorso principale '/', questa visualizza il timestamp della richiesta nel browser.

Express.js – Middleware

- Un'applicazione Express è essenzialmente una serie di chiamate a funzioni middleware.
- Un'applicazione Express può utilizzare i seguenti tipi di middleware:
  - Application-level middleware
  - Router-level middleware
  - Error-handling middleware
  - Built-in middleware
  - Third-party middleware

Application-level middleware

Express.js – Application-level middleware

- Le **application-level middleware** sono funzioni middleware che vengono associata direttamente all'applicazione Express.
- Le funzioni **app.use()** e **app.METHOD()** (dove METHOD può essere get, post, put, ecc.), permettono di associare un **application-level middleware** all'istanza della classe express **app**.
- Questo esempio mostra una funzione middleware che viene montata senza specificare il percorso.

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

app.use((req, res, next) => {
        console.log('Time:', Date.now());
        next();
});
```

- La funzione viene eseguita ogni volta che l'applicazione riceve una richiesta. Ovviamente tenete sempre in considerazione la posizione in cui inserite il codice.

Express.js – Application-level middleware

- Questo esempio mostra una funzione middleware montata sul percorso /user/:id. La funzione viene eseguita per qualsiasi tipo di richiesta HTTP sul percorso /user/:id

```
app.use('/user/:id', (req, res, next) => {
    console.log('Request Type:', req.method);
    next();
});
```

- Questo esempio mostra una route e la sua HANDLER (sistema middleware). La funzione gestisce le richieste GET al percorso /user/:id.

```
app.get('/user/:id', (req, res, next) => {
    res.send('USER');
});
```

Express.js – Application-level middleware

- Si possono caricare anche un insieme di funzioni middleware.
- L'esempio sotto illustra un sub-stack di middleware che stampano le informazioni di richiesta per qualsiasi richiesta HTTP al percorso /user/:id

```
app.use('/user/:id', (req, res, next) => {
        console.log('Request URL:', req.originalUrl);
        next();
}, (req, res, next) => {
        console.log('Request Type:', req.method);
        next();
} ..... );
```

Express.js – Application-level middleware

- Gli handler di route consentono di definire più route per un percorso.
- L'esempio seguente definisce due route per le richieste GET al percorso /user/:id.

```
app.get('/user/:id', (req, res, next) => {
        console.log('ID:', req.params.id);
        next();
}, (req, res, next) => {
        res.send('User Info');
});

// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', (req, res, next) => {
        res.send(req.params.id);
});
```

- La seconda route può essere creata (non si presentano errori), ma non verrà mai chiamata perché la prima route termina il ciclo richiesta-risposta.

Express.js – Application-level middleware

- Per saltare il resto delle funzioni middleware dello stack, si chiama **next('route').** Questa funzione passa il controllo alla route successiva.
- **NOTA:** next('route') funziona solo nelle funzioni middleware che sono state caricate usando **app.METHOD()** o **router.METHOD()**.
- Questo esempio mostra un sub-stack di middleware che gestisce le richieste GET al percorso /user/:id

```
app.get ('/user/:id', (req, res, next) => {
        if (req.params.id === '0') next('route') ; /* if the user ID is 0, skip to the next
        route*/
        else next() ; // otherwise pass the control to the next middleware in the stack
}, (req, res, next) => {
        res.send('regular') ; // send 'regular'
});

app.get('/user/:id', (req, res, next) => {
        res.send('special') ; // handler for the /user/:id path, which sends 'special'
});
```

Express.js – Application-level middleware

- I middleware possono anche essere dichiarati in un array per poter essere riutilizzati.
- Questo esempio mostra un array con un sub-stack di middleware che gestisce le richieste GET al percorso /user/:id

```
function logOriginalUrl (req, res, next) {
        console.log('Request URL:', req.originalUrl);
        next();
};

function logMethod (req, res, next) {
        console.log('Request Type:', req.method);
        next():
};

const logStuff = [logOriginalUrl, logMethod]

app.get('/user/:id', logStuff, (req, res, next) => {
        res.send('User Info');
});
```

Router-level middleware

Express.js – Router-level middleware

- I **router-level middleware** funzionano come le application-level middleware, tranne che per il fatto che sono legato a un'istanza di **express.Router().**

  const app = express(); ⟶ **const router = express.Router()**

- Il router-level middleware viene associato a **router** (istanza di express.Router()) utilizzando le funzioni **router.use()** e **router.METHOD()**.

- Possiamo replicare il sistema di middleware mostrato prima per l'application-level middleware, utilizzando il router-level middleware

```
const express = require('express');
const router = express.Router();

// Una funzione middleware montata senza specificare il PATH.
router.use((req, res, next) => {
    console.log('Time:', Date.now());
    next();
});
```

Express.js – Router-level middleware

```
/* Un sub-stack di middleware che mostra le iinformazioni dell'oggetto req per
ogni tipo di richiesta HTTP al percorso /user/:id path */
router.use('/user/:id', (req, res, next) => {
        console.log('Request URL:', req.originalUrl);
        next();
}, (req, res, next) => {
         console.log('Request Type:', req.method);
        next();
} ..... );
.....
module.exports = router;
```

- Come abbiamo visto nella scorsa lezione, questo codice verrà poi importato e montato nel file principale dell'applicazione express per essere utilizzato.

Express.js – Router-level middleware

- Per saltare il resto delle funzioni middleware del router, si chiama **next('router')** per passare il controllo all'esterno dell'istanza del router.
- Questo esempio mostra un sub-stack di middleware che gestisce le richieste GET al percorso /user/:id

adminRouter.js

```
const express = require('express');
const router = express.Router();

/* Se necessario abbandoniamo il router e saltiamo tutti i middleware che gli appartengono */
router.use((req, res, next) => {
    if (!req.headers['authorization'])
        return next('router') ;
    next();
});

router.get('/user/:id', (req, res) => {
    res.send('hello, admin!');
});

module.exports = router;
```

Express.js – Router-level middleware

index.js

```
const express = require('express');
const app = express();
const adminRouter = require('./adminRouter');

/* Per ogni richiesta che inizia con /admin, si verifica se l'utente è autorizzato
controllando l'header, in caso non si abbia il permesso di accesso si restituisce
il codice di stato 401 */
app.use('/admin', adminRouter, (req, res) => {
    res.sendStatus(401);
});
```

Built-in middleware

Express.js – Built-in middleware

- Express dispone di diverse funzioni middleware integrate.
- **express.json** analizza il corpo delle richieste in arrivo con payload JSON (Content-Type: application/json) e lo converte in un oggetto JavaScript accessibile tramite req.body. **NOTA:** Disponibile da Express 4.16.0 in poi.

```
app.use(express.json());
```

- **express.static** per gestire i file statici, quali immagini, file CSS e file JavaScript.

```
express.static(root, [options])
```

L'argomento root specifica il nome della directory che contiene le risorse statiche.
Ad esempio, per gestire le immagini, i file CSS e i file JavaScript nella cartella denominata **public**

```
app.use(express.static('public'));
```

Ora è possibile accedere ai file presenti nella directory public

http://localhost:3000/images/image.jpg

http://localhost:3000/css/style.css

http://localhost:3000/js/app.js

Third-party middleware

Express.js – Third-party middleware

- I **third-party middleware** si utilizzano per aggiungere funzionalità alle applicazioni. Express. https://expressjs.com/en/resources/middleware.html

1. Si installa il modulo Node.js per la funzionalità richiesta.
   Esempio **npm install body-parser**
2. Si monta nell'applicazione a livello di applicazione o di router

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');

app.use(bodyParser.json());
app.post('/api/data', (req, res) => {
    console.log(req.body);
    res.send('Received JSON data');
});
```

- Il middleware **bodyParser.json()** analizza il corpo della richiesta solo se Content-Type è application/json. Popola req.body con il contenuto convertito in oggetto JavaScript. Se il corpo è vuoto, il tipo non è JSON, o c'è un errore nel parsing → req.body sarà un oggetto vuoto {} o verrà lanciato un errore.
- **A partire da Express 4.16.0 sostituito da app.use(express.json());**

Error-handling middleware

Express.js – Catching Errors

- **Error Handling** si riferisce al modo in cui Express cattura ed elabora gli errori che si verificano sia in modo sincrono che asincrono.
- Express viene fornito con un gestore di errori predefinito, quindi non è necessario scriverne uno proprio (anche se si può fare).
- È importante assicurarsi che Express catturi e gestisca tutti gli errori che si verificano durante l'esecuzione di handler di route e middleware.

- Gli errori che si verificano nel codice **sincrono** all'interno di handler di route e middleware non richiedono lavoro aggiuntivo.
- Infatti se il codice sincrono lancia un errore, Express lo cattura e lo elabora senza mandare in crash il server.

```
app.get('/', (req, res) => {
    throw new Error('BROKEN'); // Express lo cattura e lo elabora da solo
});
```

Express.js – Catching Errors

- Invece per gli errori restituiti da funzioni **asincrone**, invocate da handler di route e middleware, occorre passarli alla funzione **next()** per farli catturare ed elaborare da Express.

```
const fs = require('fs'); // importiamo il file system module

app.get('/', (req, res, next) => {
    fs.readFile('/file-does-not-exist',  (err, data) => {
        if (err) {
            next(err); // passiamo l'errore a Express
        } else {
            res.send(data);
        }
    });
});
```

Express.js – Catching Errors

- A partire da **Express 5**, gli handler di route e i middleware che restituiscono una Promise chiamano automaticamente **next(valore)** quando viene richiamato reject(error) o viene generato un errore con throw

- Se getUserById genera un errore o invoca reject, next sarà chiamato con l'errore lanciato o con il valore di reject.

```
app.get('/user/:id', async (req, res, next) => {
        const user = await getUserById(req.params.id);
        res.send(user);
});
```

- Se si passa qualcosa alla funzione next() (eccetto la stringa "route" e "router"), Express considera la richiesta corrente come un errore e salta tutte le altre funzioni di routing e middleware che non gestiscono gli errori.

Express.js – Catching Errors

- È necessario catturare gli errori che si verificano nel codice asincrono invocato dagli handler di route o dai middleware e passarli a Express per l'elaborazione.

```
app.get('/', (req, res, next) => {
        setTimeout(() => {
                try {
                        throw new Error('BROKEN')
                } catch (err) {
                        next(err)
                }
        }, 100);
});
```

- L'esempio precedente utilizza un blocco try... catch per catturare gli errori nel codice asincrono e passarli a Express. Se il blocco try... catch fosse omesso, Express non catturerebbe l'errore.
- Non è necessario con async/await o Promise in Express 5.

Express.js – Default error handler

- Express è dotato di un gestore di errori integrato che si occupa di tutti gli errori che si possono incontrare nell'applicazione.
- Questa funzione middleware predefinita di gestione degli errori viene aggiunta alla fine dello stack di funzioni middleware.
- Se si passa un errore a next() e non lo si gestisce con un gestore di errori personalizzato, verrà gestito dal gestore di errori integrato. L'errore verrà scritto al client con la traccia dello stack.
- Quando viene scritto un errore, alla risposta vengono aggiunte diverse informazioni tra cui:
  - Il **codice di stato**. Se questo valore è al di fuori dell'intervallo 4xx o 5xx, sarà impostato a 500.
  - Il **messaggio di errore.** Viene impostato in base al codice di stato.
  - Il body sarà l'HTML del messaggio con codice di stato se in ambiente di produzione, altrimenti sarà **err.stack**.
- **err.stack** è una stringa che descrive il punto del codice in cui l'errore è stato istanziato.
- La prima riga stampata in console è formattata come <nome della classe di errore>: <messaggio di errore>, ed è seguita da una serie di frame di stack (ogni riga inizia con "at "). Ogni frame descrive una chiamata all'interno del codice che ha portato alla generazione dell'errore.

Express.js – Default error handler

- Se si chiama next() con un errore dopo aver iniziato a scrivere la risposta (per esempio, se si verifica un errore durante lo streaming della risposta al client), il gestore di errori predefinito di Express chiude la connessione e fallisce la richiesta.

- Pertanto, quando si aggiunge un gestore di errori personalizzato, si deve delegare al gestore di errori predefinito di Express, quando gli header sono già stati inviati al client

```
function errorHandler (err, req, res, next) {
        if (res.headersSent) {
                return next(err);
        }
        res.status(500).send(`Errore del server: ${err.message}`);
};
```

Express.js – Writing error handlers

- Gli **error-handling middleware** si definiscono allo stesso modo delle altre funzioni middleware, con la differenza che le funzioni di gestione degli errori hanno quattro argomenti invece di tre: (err, req, res, next).

```
app.use((err, req, res, next) => {
        console.error(err.stack);
        res.status(500).send('Something broke!');
});
```

- Si definisce il middleware di gestione degli errori per ultimo, dopo le altre chiamate ad app.use() e alle route
- Le risposte di una funzione middleware possono essere in qualsiasi formato, come una pagina di errore HTML, un semplice messaggio o una stringa JSON.
- Per motivi organizzativi è possibile definire diverse funzioni middleware di gestione degli errori, proprio come si farebbe con le normali funzioni middleware.

```
.....
app.use(notFoundErrorsHandler)
app.use(badRequestErrorsHandler)
app.use(errorHandler)
.....
```