Tecnologie e applicazioni web

Express

Filippo Bergamasco (filippo.bergamasco@unive.it)
http://www.dais.unive.it/~bergamasco/
DAIS - Università Ca'Foscari di Venezia
Anno accademico: 2017/2018

Express.js

Express è un modulo Node.js che fornisce un'infrastruttura middleware semplice e robusta per la creazione di applicazioni web

Permette di risolvere in modo strutturato i seguenti problemi:

- Avere molteplici funzioni che leggono/modificano in sequenza gli oggetti request e response
- Gestire il **routing** tra gli endpoint e le API

```
else if( req.url.search( "/messages" )!=-1 && req.method == "GET" ) {

    var query = url.parse( req.url, true /* true=parse query string*/).query;
    console.log(" Query: ".red + JSON.stringify(query));

    var filter = {};
    if( query.tags ) {
        filter = { tags: {$all: query.tags } };
    }
    console.log("Using filter: " + JSON.stringify(filter) );

    query.skip = parseInt( query.skip || "0" ) || 0;
    query.limit = parseInt( query.limit || "20" ) || 20;

    message.getModel().find( filter ).skip( query.skip ).limit( query.limit ).then( (documents) => {
        return respond( 200, documents );
    }).catch( (reason) => {
        return respond(404, { error: true, errormessage: "DB error:"+reason });
    })
}
```

Routing

```
else if( req.url.search( "/messages" )!=-1 && req.method == "GET" ) {

    var query = url.parse( req.url, true /* true=parse query string*/).query;
    console.log(" Query: ".red + JSON.stringify(query));

    var filter = {};
    if( query.tags ) {
        filter = { tags: {$all: query.tags } };
    }
    console.log("Using filter: " + JSON.stringify(filter) );

    query.skip = parseInt( query.skip || "0" ) || 0;
    query.limit = parseInt( query.limit || "20" ) || 20;

    message.getModel().find( filter ).skip( query.skip ).limit( query.limit ).then( (documents) => {
        return respond( 200, documents );
    }).catch( (reason) => {
        return respond(404, { error: true, errormessage: "DB error:"+reason });
    })
}
```

```
else if( req.url.search( "/messages" )!=-1 && req.method == "GET" ) {

    var query = url.parse( req.url, true /* true=parse query string*/).query;
    console.log(" Query: ".red + JSON.stringify(query));

    var filter = {};
    if( query.tags ) {
        filter = { tags: {$all: query.tags } };
    }
    console.log("Using filter: " + JSON.stringify(filter) );

    query.skip = parseInt( query.skip || "0" ) || 0;
    query.limit = parseInt( query.limit || "20" ) || 20;

    message.getModel().find( filter ).skip( query.skip ).limit( query.limit ).then( (documents) => {
        return respond( 200, documents );
    }).catch( (reason) => {
        return respond(404, { error: true, errormessage: "DB error:"+reason });
    })
}
```

Query parsing middleware

```
else if( req.url.search( "/messages" )!=-1 && req.method == "GET" ) {

    var query = url.parse( req.url, true /* true=parse query string*/).query;
    console.log(" Query: ".red + JSON.stringify(query));

    var filter = {};
    if( query.tags ) {
        filter = { tags: {$all: query.tags } };
    }
    console.log("Using filter: " + JSON.stringify(filter) );

    query.skip = parseInt( query.skip || "0" ) || 0;
    query.limit = parseInt( query.limit || "20" ) || 20;

    message.getModel().find( filter ).skip( query.skip ).limit( query.limit ).then( (documents) => {
        return respond( 200, documents );
    }).catch( (reason) => {
        return respond(404, { error: true, errormessage: "DB error:"+reason });
    })
}
```

MongoDB filter middleware

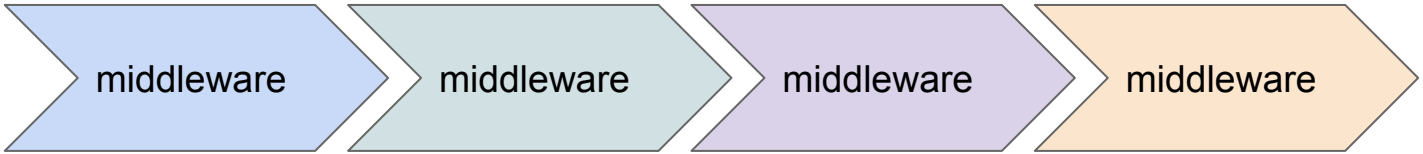
```
else if( req.url.search( "/messages" )!=-1 && req.method == "GET" ) {

    var query = url.parse( req.url, true /* true=parse query string*/).query;
    console.log(" Query: ".red + JSON.stringify(query));

    var filter = {};
    if( query.tags ) {
        filter = { tags: {$all: query.tags } };
    }
    console.log("Using filter: " + JSON.stringify(filter) );

    query.skip = parseInt( query.skip || "0" ) || 0;
    query.limit = parseInt( query.limit || "20" ) || 20;

    message.getModel().find( filter ).skip( query.skip ).limit( query.limit ).then( (documents) => {
        return respond( 200, documents );
    }).catch( (reason) => {
        return respond(404, { error: true, errormessage: "DB error:"+reason });
    })
}
```

Logging middleware

```
else if( req.url.search( "/messages" )!=-1 && req.method == "GET" ) {

    var query = url.parse( req.url, true /* true=parse query string*/).query;
    console.log(" Query: ".red + JSON.stringify(query));

    var filter = {};
    if( query.tags ) {
        filter = { tags: {$all: query.tags } };
    }
    console.log("Using filter: " + JSON.stringify(filter) );

    query.skip = parseInt( query.skip || "0" ) || 0;
    query.limit = parseInt( query.limit || "20" ) || 20;

    message.getModel().find( filter ).skip( query.skip ).limit( query.limit ).then( (documents) => {
        return respond( 200, documents );
    }).catch( (reason) => {
        return respond(404, { error: true, errormessage: "DB error:"+reason });
    })
}
```

Rendering

Middleware



Il middleware è una funzione che prende in input oggetti request e response, opera sui loro valori, e invoca la funzione di middleware successiva nella pipeline (oppure conclude la response)

```
function (req, res, next) { }
```

Middleware built-in

Express contiene una serie di middleware built-in per gestire i casi d'uso più frequenti:

- Parsing di stringhe JSON e dati da form
- Compressione dei dati
- Inserimento e parsing dei cookies
- Gestione delle sessioni
- Invio di files e pagine HTML statiche

Routing

Le regole di routing specificano come gli endpoint (URLs) si mappano con le rispettive pipeline di middleware

Vantaggi:

- Gestione strutturata dei path invece di enormi blocchi if - else if - else
- Uso di espressioni regolari per il matching di una route e l'estrazione automatica di parametri

Definizione del routing

`app.<method>(<path>,[<handlers>])`

- app è un istanza del modulo Express
- `<method>` è un metodo HTTP: get, post, etc.
- `<path>` è la regular expression che indica l'endpoint
- `[<handlers>]` è un array di funzioni middleware che verranno eseguite in sequenza se l'URL della richiesta soddisfa il `<path>`. Molto spesso l'array è composto da una sola funzione

Definizione del routing

```
app.get('/users/:userId/books/:bookId', function (req, res) {
  res.send(req.params)
})
```

Il `<path>` può prevedere dei segmenti variabili che vanno parsati come parametri (:userId, :bookId). Express gestisce automaticamente l'operazione inserendo ciascun parametro all'interno della request

Funzioni middleware globali

Alcune funzioni di middleware possono essere inserite nella pipeline indipendentemente dal metodo HTTP o dal path dell'endpoint

Ex:

```
app.use( express.bodyParser() )

app.use( express.methodOverride() )
```

Attenzione all'ordine di inserimento della pipeline

Middleware gestione errori

E' possibile definire dei middleware "speciali" che si occupano di gestire eventuali errori che avvengono durante il routing o nei middleware precedenti

Sono funzioni che possiedono 4 argomenti, e vengono solitamente installate alla fine della pipeline:

```
function (err, req, res, next) { }
```

Middleware gestione errori

Le funzioni per la gestione degli errori vengono invocate:

- Se è stata lanciata un'eccezione (Le eccezioni vanno però evitate in un contesto asincrono)
- Se un middleware invoca next(..) senza passare la request e la response, ma solo un oggetto che descrive l'errore **(metodo consigliato)**

Gestione pagine statiche

Uno dei middleware built-in più usati è per l'invio di pagine statiche ai client:

- Si inseriscono i files statici all'interno di una o più directory <dir1>, <dir2>, etc
- Si inserisce il middleware nella pipeline al livello desiderato:

```
app.use(express.static('<dir1>'));
app.use(express.static('<dir2>'));
```