JavaScript runtime built on Chrome's V8 JavaScript engine

Node.js

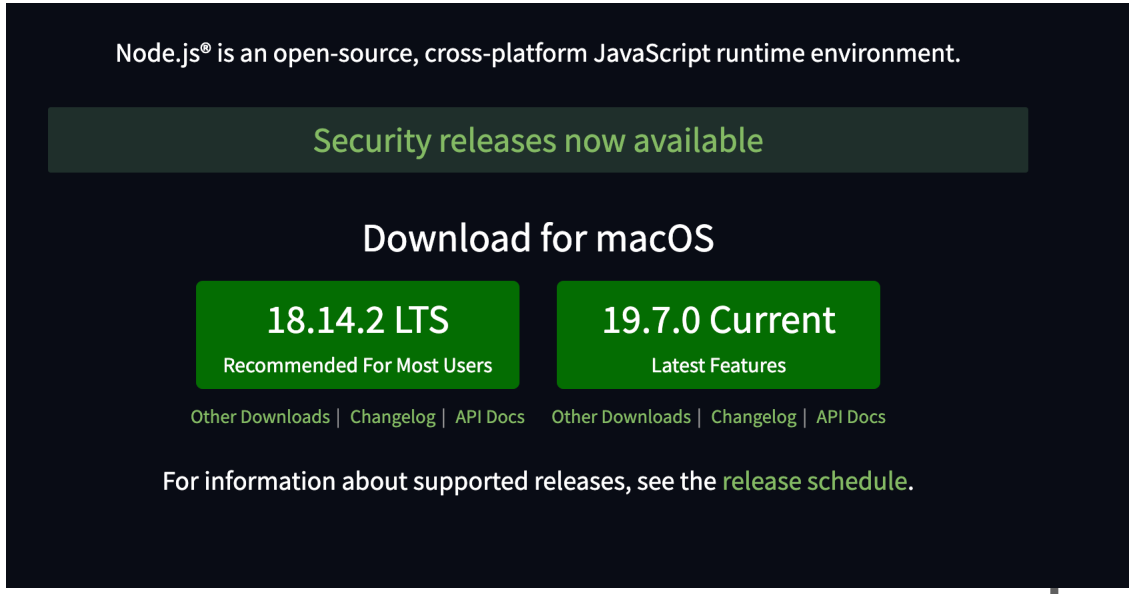
- Piattaforma software cross-platform
  - Nota: non è un web server e neanche un linguaggio
- Realizzato su Google Chrome V8 javascript engine
  - Esegue codice JavaScript server side
- Single-threaded
- **Event-driven** architecture
- Asynchronous
- Non blocking I/O model

Web Server

- **Node.js non è un web server**: è una runtime environment che permette di eseguire codice JavaScript lato server.
  - Non funziona come Apache: no config file
- **Può essere usato per sviluppare un web server**
  - Si può scrivere un server HTTP con l'utilizzo delle librerie fornite

Installazione

- Scaricare ed eseguire l'installer di Node.js
  - https://nodejs.org/en/
- Verificare l'istallazione
  - `node -v`



Esercizio 1

Utilizzare l'engine JavaScript di Node.js per eseguire codice senza l'utilizzo di alcun web server

- Creare il file app.js
  - Utilizzare la console JS
  - `console.log("Hello World!");`
- Eseguire
  - `node app.js`

Esercizio 2

- Creare un server HTTP con Node.js

Esercizio 2

```
const http = require('http');
```

- **“Require” permette di includere moduli**
- In questo caso il modulo built-in *“http”* che permette il trasferimento di pacchetti HTTP

Esercizio 2

```
http.createServer((req, res)=>{});
```

- Parametri:
  - **Req**: richiesta
  - **Res**: risposta
- Documentazione: https://nodejs.org/api/http.html

Esercizio 2

```
5   const http = require('http');
6   const server = http.createServer( requestListener: (req : IncomingMessage ,res : ServerResponse )=>{
7       res.write( chunk: 'Hello world!'); //write a response to the client
8       res.end(); //end the response
9   });
10  server.listen( port: 8080);
```

Esercizio 2

- Eseguire il file

```
node app.js
>
```

- Il server è in attesa di ricevere richieste dal client.
- Come si invia una richiesta al server?

Esercizio 2

- Dal browser
- Il server è in ascolto su localhost:8080

Esercizio 2

- Se vogliamo che il risultato sia visualizzato in HTML, allora bisogna settare il **Content-Type**

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('Hello World!');
  res.end();//to close the response
}).listen(8080);
```

- 200 = The request has succeeded

Esercizio 2

- Entrambi gli oggetti `res` e `req` hanno delle proprietà
- Come, per esempio:
  - `req.url;`
  - `req.method;`

Moduli

- HTTP
  - Creazione server e utilizzo del protocollo
- URL
  - Da oggetti a Url e viceversa
- PATH
  - Lavora con i percorsi reali della macchina
- FS
  - Creazione, copia, cancellazione, … di cartelle e file
- UTIL
  - IsArray, format, ...
- NET
  - Per lavorare con la rete a più basso livello

Moduli Custom (CommonJS)

- Un modulo custom è un file JS che implementa alcune funzioni e le espone tramite un oggetto **exports**
- Le funzioni appese all'oggetto exports diventano pubbliche. Tutte le altre restano private
- Importare un modulo:
  - Specificando solo il nome del modulo da importare, questo verrà cercato nella cartella `node_modules`
  - Per importare un modulo custom bisogna specificare il path relativo (o assoluto)

Moduli Custom

```
//stringmodule.js
const concat = (a,b)=>{return a + b;}
const upper = (a)=>{return a.toUpperCase();}
const fruit = ()=>{return 'b' + 'a' + + 'u' + 'a';}

exports.concat = concat;
exports.fruit = fruit;

//app.js

const sm = require('./stringmodule');
console.log(sm.concat("hello","world")); //helloworld
console.log(sm.fruit()); //baNaNa
console.log(sm.upper("ciao")); //error
```

exports VS module.exports

```
exports.a = 'A';
module.exports.b = 'B';
console.log(exports === module.exports);
```

```
Module {
  id: '.',
  exports: { a: 'A', b: 'B' },
…
true
```

exports VS module.exports

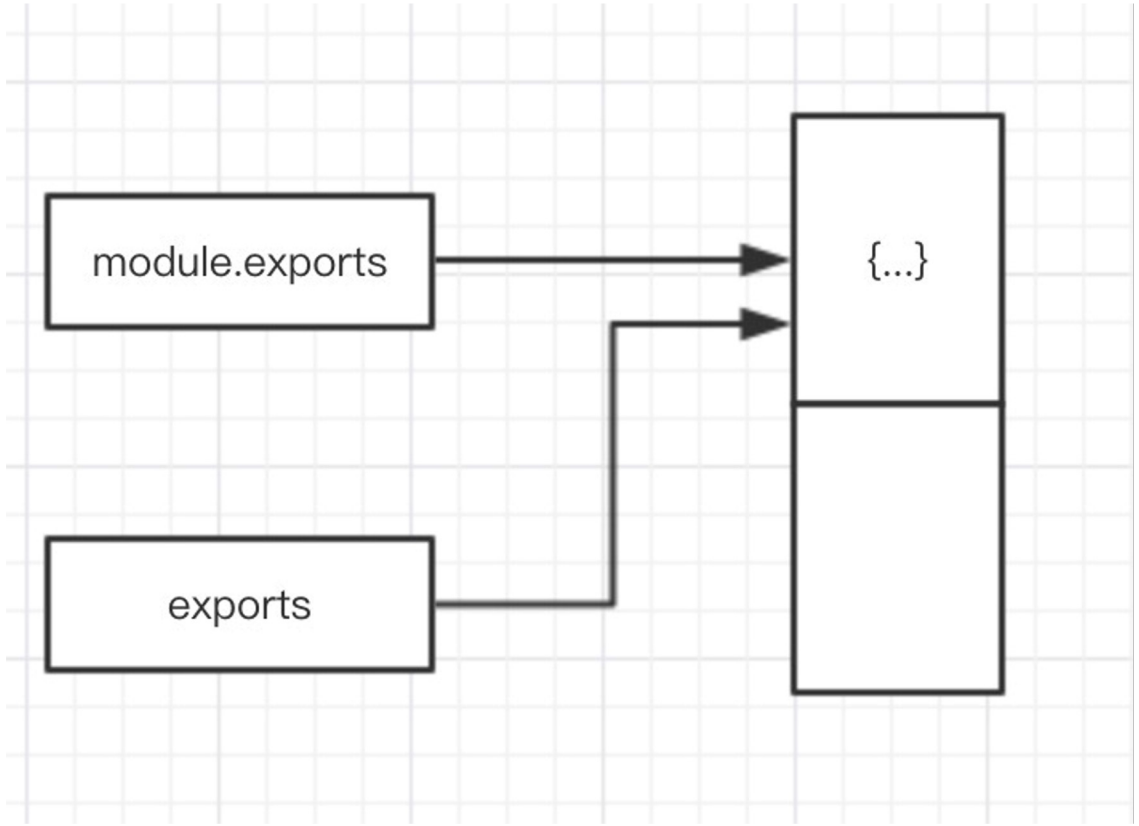
Using module.exports:

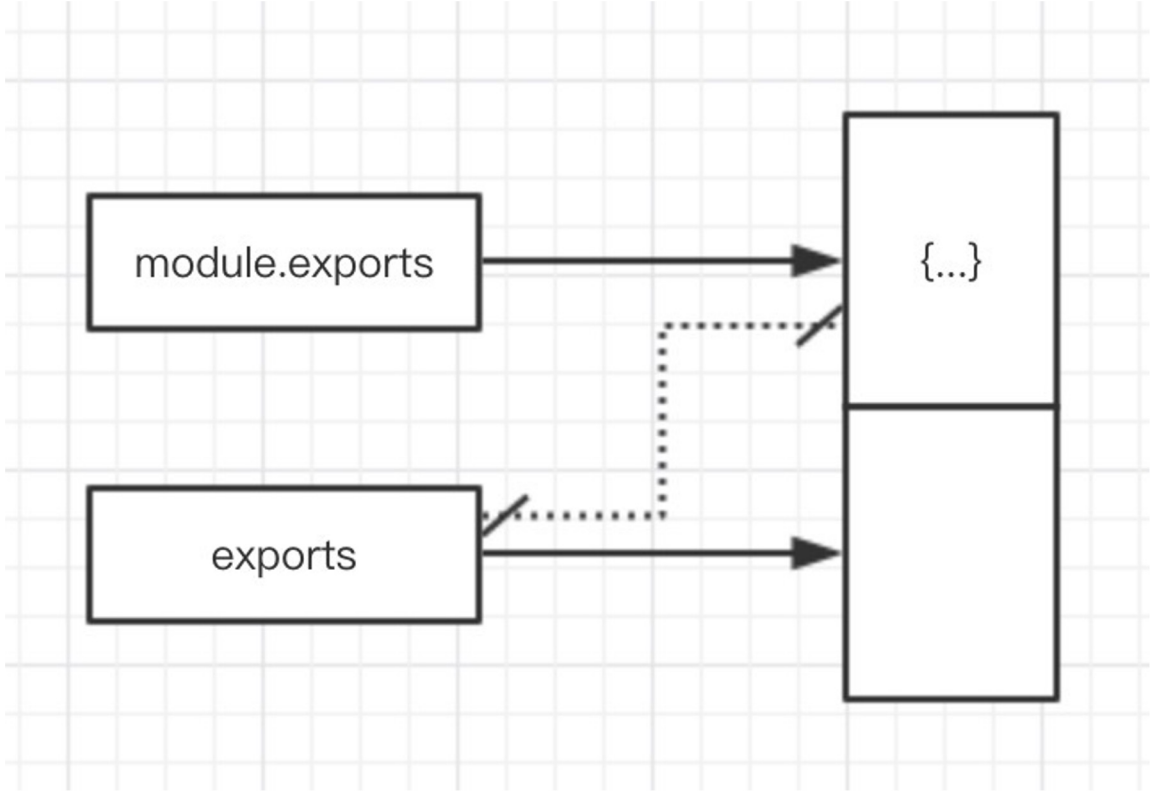
```
module.exports = {
  greet: function (name) {
    console.log(`Hi ${name}!`);
  },
  farewell: function() {
    console.log('Bye!');
  }
}
```

Using exports:

```
exports.greet = function (name)
{
  console.log(`Hi ${name}!`);
}
exports.farewell = function() {
  console.log('Bye!');
}
```

exports VS module.exports





exports VS module.exports

- Module.exports
  - Require restituisce sempre module.exports
  - Può essere usato in **append** o in **replace** (in tal caso export non punterà più allo stesso oggetto)
- Exports
  - E' una variabile fornita per abbreviare il codice
  - Da usare solo in **append**
  - Non viene restituita da require

```
exports = module.exports = {}
```

Esercizio 3

- Scrivere un modulo “routes” che possa essere utilizzato come segue

```
const http = require('http');

const routes = require('./routes');

const server = http.createServer(routes.handler);

server.listen(3000, ()=>{
   console.log("Server in ascolto su http://localhost:3000")
});
```

GET sulla rotta /

POST sulla rotta /message

deve stampare sulla console il messaggio inviato

Esercizio 3

- Come faccio a fare il parsing del body?
- Dato che il body viene inviato in chunk, devo riassemblarlo:

```
const body = [];
req.on('data', chunk => {
   body.push(chunk);
});
```

- Alla chiusura della request lo converto in stringa:

```
req.on('end', () => {
   const parsedBody = Buffer.concat(body).toString();
});
```

Module: ECMAScript

- I moduli ECMAScript sono un formato standard ufficiale per impacchettare il codice. I moduli sono definiti utilizzando istruzioni di import ed export.

```
//stringmodule.mjs
const concat = (a,b)=>{return a + b;}
const upper = (a)=>{return a.toUpperCase();}
const fruit = ()=>{return 'b' + 'a' +  +'u' + 'a';}

export {concat, fruit};


//app.mjs
import * from './stringmodule.js'

console.log(concat("hello","world")); //helloworld
console.log(fruit()); //baNaNa
console.log(upper("ciao")); //error
```

Module: ECMAScript

- per usare moduli ES6 ho tre possibilità:
  - inserire in package.json "type": "module"
  - usare l'estensione .mjs per specificare l'uso di moduli ES6 in tutti i file js
  - lanciare node con l'opzione --input-type=module

Modulo Express

- Express
  - “Fast, unopinionated, minimalist web framework for Node.js”
  - Può svolgere gli stessi compiti che svolge il codice appena scritto, ed altri
  - http://expressjs.com/

Esercizio 4

- Creare un webserver con Express in modo che quando l'utente visiti la root del sito venga restituita la stringa `Hello World!`

Esercizio 4

- Creare una cartella `Esercizio04`
- Posizionarsi con la shell nella cartella appena creata ed eseguire il comando `npm init`
  - - Verrà creato il file **package**.json, che descrive tutte le dipendenze del progetto.
- Lasciare le impostazioni di default eccetto l'entry point, che chiameremo `app.js`.

Esercizio 4

- Aggiungere express al progetto

```
npm install express
```

- Cambiamenti
  1. `package.json`: nell'elenco delle dipendenze è stato aggiunto express
  2. `package-lock.json`: aggiunto questo file che contiene un elenco dettagliato di TUTTE le dipendenze.
  3. `node_modules`: cartella che contiene il codice dei moduli importanti

Esercizio 4

- Importare Express con il comando
  `const express = require('express');`
- Specificando solo il nome del modulo da importare, questo verrà cercato nella cartella `node_modules`
- Per importare un modulo custom specificare il path relativo

Esercizio 4

- Creare l'applicazione express

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

- Specificare la gestione della rotta /

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

Esercizio 4

- Infine mettere il server in ascolto sulla porta 3000
- ```
  app.listen(3000, ()=> {
     console.log('Listening on port 3000');
  });
  ```
  ```

Esercizio 4

- Cosa succede visitando la rotta /asw ?

```
Cannot GET /asw
```

Esercizio 4

- Gestire gli errori **404**
- Aggiungere **dopo** la gestione della rotta “/”, ma prima dell’avvio del server il seguente codice

```
app.use((req, res, next)=>{
   res.setHeader('Content-Type', 'text/plain’);
   res.status(404).send('Ops... Pagina non
trovata’);
});
```

Esercizio 4.1: Restituire json

- Dato il file **colors.json** presente nei file delle esercitazioni, restituire il suo contenuto in corrispondenza di richieste al percorso /colors

Esercizio 4.1: Restituire json

- Importare i dati. Si può fare con il comando require

```
const data = require('./colors.json’);
```

- Successivamente, definire la rotta /colors e il relativo handler

```
app.get('/colors', (req, res)=>{});
```

Esercizio 4.1: Restituire json

- Due possibili alternative: send e json
- Nel primo caso, è necessario definire l'intestazione della risposta e poi mandare i dati sotto forma di json

```
res.header("Content-Type",'application/json');
res.send(JSON.stringify(data));
```

- Alternativamente Usando direttamente il metodo json, non è necessario specificare il tipo di contenuto

```
res.json(data);
```

Esercizio 4.2: Restituire un file html

- Aggiungere il file **contacts.html** presente nei file delle esercitazioni nella cartella www (da creare).
- Restituire il file in corrispondenza di richieste al percorso /contacts

Esercizio 4.2: Restituire un file html

- È possibile usare la funzione sendFile.
- NB: Richiede un path assoluto

```
app.get('/contacts', (req, res)=>{
    res.sendFile(__dirname  + '/www/contacts.html’);
});
```

Esercizio 4.2 bis: Restituire un file html

- Nell'esercizio precedente, il codice html e css erano uniti nello stesso file (bad practice).
- Ma cosa succede quando sono separati? E magari la pagina html include anche 3 script javascript?
- È necessario specificare una rotta per ogni file statico?

Esercizio 4.2 bis: Restituire un file html

- Solitamente si definisce una cartella **public**, in cui vengono inseriti tutti i file statici (css, js, immagini, ecc…).
- Successivamente, con il seguente comando, è possibile gestirli tutti.

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

Esercizio 4.2 bis: Restituire un file html

- Creare una cartella public e al suo interno:
  - inserire il file `contacts-no-css.html`
  - Creare una cartella `css` con all'interno il file `style.css`
- Aggiungere il comando

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

- Visitare la rotta /contacts-no-css.html

Esercizio 4.3: Leggere parametri

- Esempio con id user
  1. /user/12345
  2. /user?id=12345
- Come si gestiscono in Express?
  1. Con req.params.id usando la rotta /user/:id
  2. Con req.query.id usando la rotta /user

Esercizio 4.3: Leggere parametri

- Definire una rotta sayhello/nome che prenda come parametro un nome e restituisca in output “Hello ***Nome***!”

Esercizio 4.3: Leggere parametri

```
app.get('/sayhello/:name', (req, res)=>{
     res.send("Hello " + req.params.name + "!");
});
```

Template

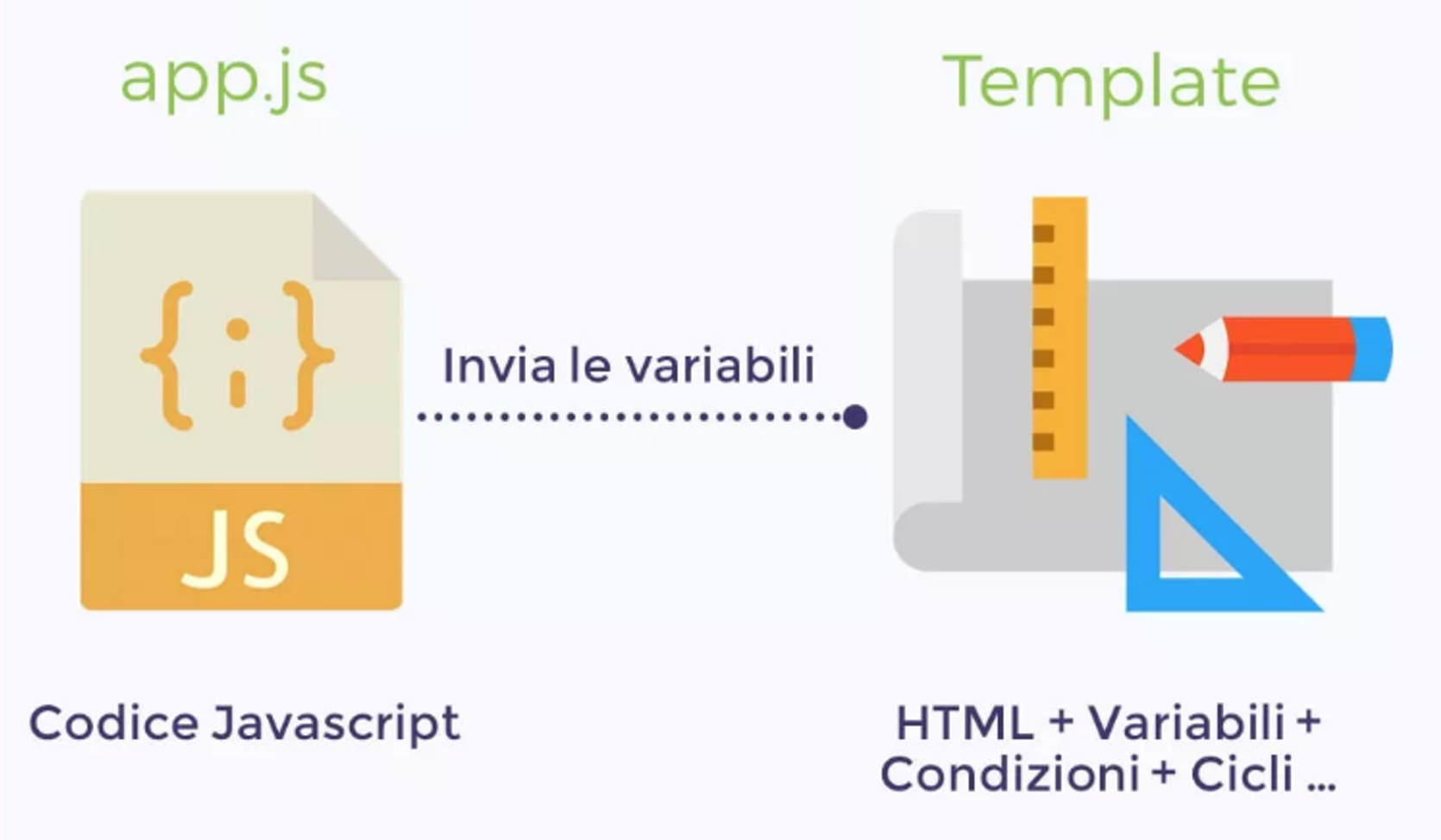
- **Problema**: logica e presentazione non sono separate

```
res.write('<!DOCTYPE html>'+
'<html>'+
'    <head>'+
'        <meta charset="utf-8" />'+
'        <title>Pagina Node.js!</title>'+
'    </head>'+
'    <body>'+
'       <p>Sono un paragrafo <strong>HTML</strong>!</p>'+
'    </body>'+
'</html>');
```

Template

- **Soluzione**: template engine
- Un template engine consente di utilizzare template statici nell'applicazione. In fase di esecuzione, l'engine sostituisce le variabili con i valori attuali e trasforma il template in un file HTML da inviare al client.

Template



https://www.nodeacademy.it/cose-ejs-template-engine-express-js/

Template

- Alcuni Template Engine
- Di default viene usato Pug (Jade)

- **Pug**: Haml-inspired template engine (formerly Jade).
- **Haml.js**: Haml implementation.
- **EJS**: Embedded JavaScript template engine.
- **hbs**: Adapter for Handlebars.js, an extension of Mustache.js template engine.
- **Squirrelly**: Blazing-fast template engine that supports partials, helpers, custom tags, and caching. Not white-space sensitive, works with any language.
- **React**: Renders React components on the server. It renders static markup and does not support mounting those views on the client.
- **h4e**: Adapter for Hogan.js, with support for partials and layouts.
- **hulk-hogan**: Adapter for Twitter's Hogan.js (Mustache syntax), with support for Partials.
- **combyne.js**: A template engine that hopefully works the way you'd expect.
- **swig**: Fast, Django-like template engine.
- **Nunjucks**: Inspired by jinja/twig.
- **marko**: A fast and lightweight HTML-based templating engine that compiles templates to CommonJS modules and supports streaming, async rendering and custom tags. (Renders directly to the HTTP response stream).
- **whiskers**: Small, fast, mustachioed.
- **Blade**: HTML Template Compiler, inspired by Jade & Haml.
- **Haml-Coffee**: Haml templates where you can write inline CoffeeScript.
- **Webfiller**: Plain-html5 dual-side rendering, self-configuring routes, organized source tree, 100% js.
- **express-hbs**: Handlebars with layouts, partials and blocks for express 3 from Barc.
- **express-handlebars**: A Handlebars view engine for Express which doesn't suck.
- **express-views-dom**: A DOM view engine for Express.
- **rivets-server**: Render Rivets.js templates on the server.
- **Exbars**: A flexible Handlebars view engine for Express.
- **Liquidjs**: A Liquid engine implementation for both Node.js and browsers.
- **express-tl**: A template-literal engine implementation for Express.
- **vuexpress**: A Vue.js server side rendering engine for Express.js.

https://expressjs.com/en/resources/template-engines.html

Esercizio 4.4: Template engine e Express

- Aggiungere Pug al progetto

```
npm install pug
```

- Impostare Pug come template engine

```
app.set('view engine', 'pug');
```

- Di default, i template vengono cercati nella cartella “views”.

Esercizio 4.4

- Definire una rotta tehello/nome che prenda come parametro un nome e renderizzi il template hello.

Esercizio 4.4

- Definire una rotta tehello/nome che prenda come parametro un nome e renderizzi il template hello.

```
app.get('/tehello/:name', (req, res)=>{
    res.render("hello", {name: req.params.name});
});
```

Esercizio 4.4 bis

- Definire una rotta conta/numero che prenda come parametro un numero e visualizzi i numeri da 0 a numero (compreso), utilizzando il template visualizza_numeri.

Esercizio 4.4 bis

- Definire una rotta conta/numero che prenda come parametro un numero e visualizzi i numeri da 0 a numero (compreso), utilizzando il template visualizza_numeri.

```
app.get('/conta/:numero', (req, res)=> {
    nums = [];
    for(let i = 0; i <= req.params.numero; i++){
                        nums.push(i);
    }
    res.render("visualizza_numeri", {numeri: nums});
});
```

Esercizio 4.5

- Definire una rotta *tehello/nome*

che, dato il titolo e l'email associati all'app, li inserisca nel template hello.pug (oltre che il nome, come già fatto prima)

Esercizio 4.5

- Modificare il template:

```
doctype html
html
	head
		title Hello World
	body
		h1 Hello, #{name}!
		h2 Welcome to #{title}!
		p Per maggiori info: #{email}
```

Esercizio 4.5

- Nuovi parametri:

```
app.locals.title = "My web site";
app.locals.email = "io@me.it";
```

- Uso del template

```
app.get('/tehello/mycontact/:name', (req, res)=>{
            res.render("hello", {
                  name: req.params.name,
                  title: app.locals.title,
                  email: app.locals.email
            });
});
```

Link utili

- https://www.nodeacademy.it/
- https://nodejs.org/dist/latest-v16.x/docs/api/index.html
- http://expressjs.com/