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
- Single-threaded
- **Event-driven** architecture
- Asynchronous
- Non blocking I/O model

JavaScript Runtime Environment

- **Node.js non è un web server**: è un runtime environment che permette di eseguire codice JavaScript fuori dal browser.
  - Non funziona come e.g. Apache HTTP Server: no config file
  - https://github.com/nodejs/node/
- **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`

Esempio 1

Utilizzare Node.js per eseguire codice JavaScript.

- Creare il file `app.js` con il seguente contenuto
  - `console.log("Hello World!");`
- Eseguire da terminale
  - `node app.js`

Nota: se l'argomento è una directory, verrà eseguito il file `index.js`

Regole di stile

JavaScript non ha delle **regole di stile** globalmente accettate.

Buone prassi:

- Essere coerenti
- Seguire delle linee guida
  - W3School: https://www.w3schools.com/js/js_conventions.asp
  - Google: https://google.github.io/styleguide/jsguide.html
  - Airbnb: https://github.com/airbnb/javascript

Moduli built-in

Node.js offre dei moduli per le funzioni comunemente utilizzate.

- 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 agli utilizzatori
- 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` (moduli installati tramite *package manager*)
  - Per importare un modulo custom bisogna specificare il path relativo (o assoluto)

Moduli Custom (Esempio 2)

```
// my-module.js
const add = (a, b) ⇒ a + b;
const sub = (a, b) ⇒ a - b;

module.exports = { add };


// index.js
const myModule = require('./my-module');

console.log(myModule.add(1, 2)); // 3
console.log(myModule.sub(1, 2)); // TypeError: myModule.sub is not a function
```

Nota: il *path* di import è relativo al file che eseguire `require()`

exports VS module.exports

`exports` è un riferimento a `module.exports`

```
1 exports.a = 'A';
2 module.exports.b = 'B';
3
4 console.log(exports === module.exports); // true
5 console.log(module.exports);             // { a: 'A', b: 'B' }
6 console.log(exports);                    // { a: 'A', b: 'B' }
7
8 exports = { c: 'C' };
9
10 console.log(exports === module.exports); // false
11 console.log(module.exports);             // { a: 'A', b: 'B' }
12 console.log(exports);                    // { c: 'C' }
```

exports VS module.exports

module.exports

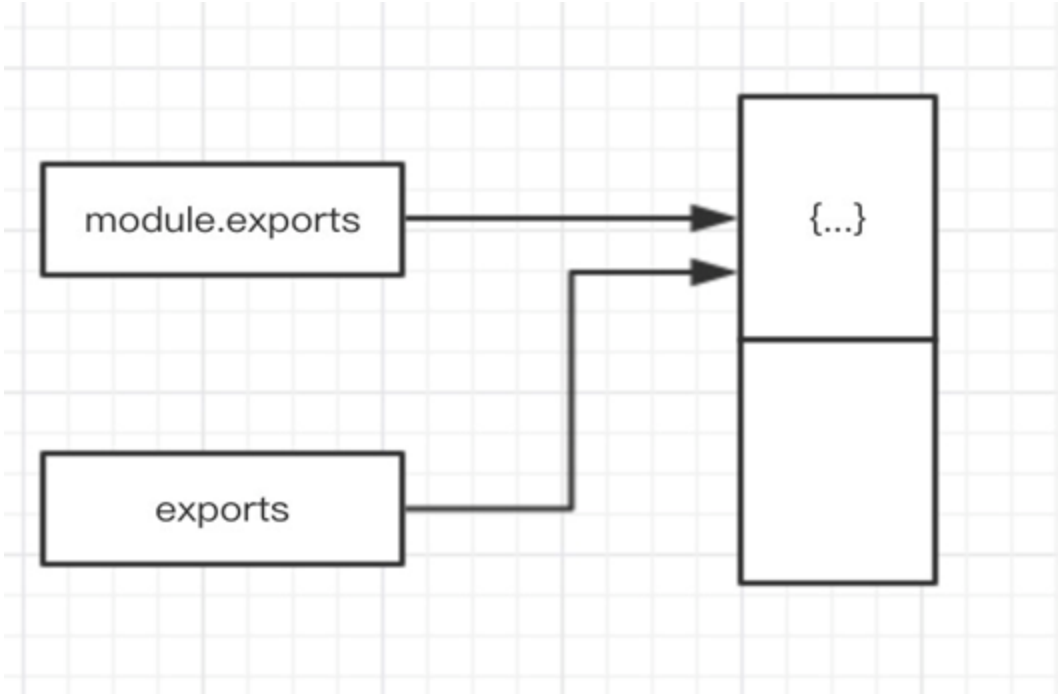
```
module.exports = {
  greet: function (name) {
    console.log(`Hi ${name}!`);
  },
  farewell: function() {
    console.log('Bye!');
  }
}
```

exports

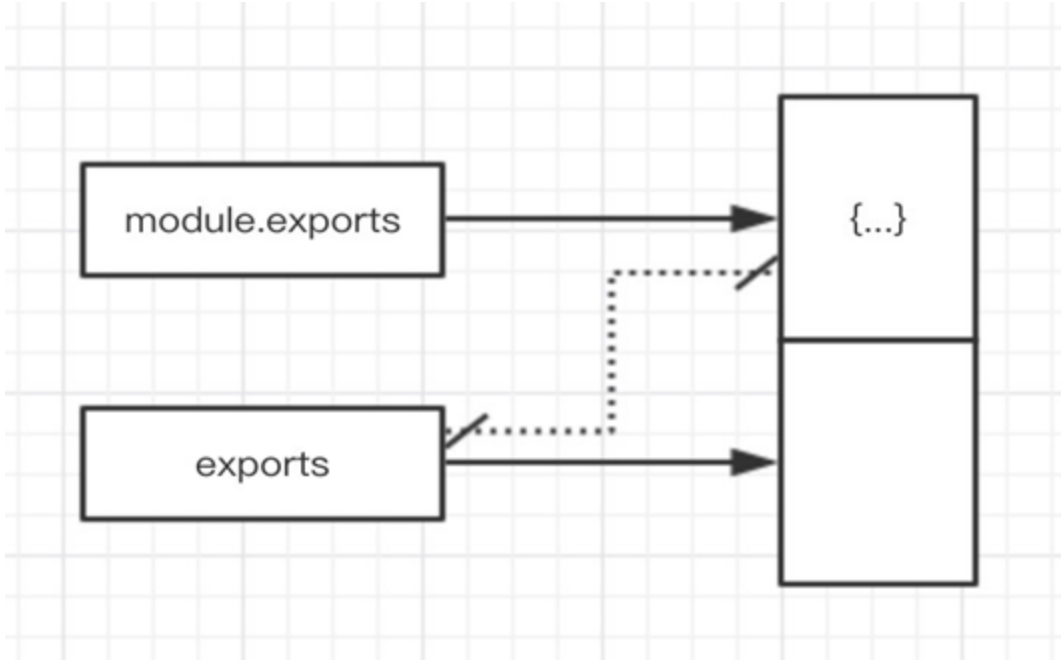
```
exports.greet = function (name) {
  console.log(`Hi ${name}!`);
}


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

exports VS module.exports

- `exports` è un riferimento a `module.exports`, costituendo un'abbreviazione
- Riassegnando il valore di `exports` viene perso il riferimento
- Ciò che viene esportato è sempre e solo `module.exports`



Situazione iniziale



Riassegno il valore di `exports`

Protocollo HTTP

HyperText Transfer Protocol (**HTTP**) è un protocollo a *livello applicativo* usato come principale sistema per la trasmissione delle informazioni sul web utilizzando un'architettura **client-server**.

Esempio 3

Creazione di un server HTTP con Node.js

Server HTTP

- Utilizzo del modulo built-in *“http”*
- Parametri della *callback*:
  - `Req`: richiesta
  - `Res`: risposta

```
1 const http = require('http');
2
3 const server = http.createServer((req, res)=> {
4   res.write('Hello World!');
5   res.end();
6 });
7
8 server.listen(8080);
```

Esempio 3

- Eseguire il file

```
node examples/example-03/app-01-simple-server.js
>
```

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

Esempio 3

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

Esempio 3

- Tramite strumenti a linea di comando
  - E.g. `curl`, `httpie`, etc.
- Tramite strumenti con interfaccia utente
  - E.g. Postman, etc.
- Realizzando il proprio *client http*
- …

Esempio 3

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

```
1  const server = http.createServer((req, res)⇒ {
2
3    // Write status code and HTTP headers
4    res.writeHead(200, {
5        'Content-Type': 'text/html',
6        'Ex2-My-Fantastic-Header': 'Hello World my header!',
7        'Request-Url': req.url
8      }
9    );
10
11   res.write(`Hello World!\n`);
12   res.write(`Used method: ${req.method}`);
13   res.end();
14 });
```

Esempio 3

- Entrambi gli oggetti `res` e `req` hanno delle proprietà
- Come, per esempio:
  - `req.url;`
  - `req.method;`
- Documentazione: https://nodejs.org/api/http.html

Esercizio 1

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

```
1 const http = require('http');
2 const routes = require('./routes');
3 const server = http.createServer(routes.handler);
```

2. Sulla rotta / con metodo GET deve servire la seguente pagina HTML

3. Sulla rotta /message con metodo POST deve ricevere il contenuto dell’input e stamparlo nella console

Esercizio 1

Suggerimento

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

```
let body = ''
req.on('data', chunk ⇒ {
    body += chunk;
});

req.on('end', () ⇒ {
    ...
});
```

NPM

Node Package Manager (**npm**) è un *gestore di pacchetti* per il linguaggio di programmazione JavaScript, predefinito per Node.js.

Consiste in:

- client da linea di comando
- database di pacchetti pubblici e privati (https://www.npmjs.com)

NPM

`npm init`: permette di creare un nuovo pacchetto npm

`npm install`: installa le dipendenze del pacchetto corrente

`npm install <nome-pacchetto>`: installa un nuovo pacchetto

`npm run <comando>`: esegue il comando specificato

NPM

Il seguente comando permette di inizializzare un pacchetto, creando il file `package.json`, che contiene i **metadati** e le **configurazioni** del progetto (e.g. versione, dipendenze, etc.)

```
> npm init

package name: (tmp)
version: (1.0.0)
entry point: (index.js)
…
```

NPM

Esempio di contenuto del file `package.json`

```
{
  "name": "tmp",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Moduli Custom (ECMAScript)

Node.js supporta i moduli **ECMAScript**, che sono un formato standard ufficiale per impacchettare il codice. I moduli sono definiti mediante utilizzando istruzioni di `import` ed `export`.

A differenza dei moduli **CommonJS**, che è la tipologia predefinita, i moduli **ECMAScript** possono essere abilitati mediante:

- `"type": "module"` in `package.json`
- usare l'estensione `.mjs` per specificare l'uso di moduli ECMAScript in tutti i file js
- eseguire `node` con l'opzione `--input-type=module` oppure `--experimental-default-type=module`

Moduli ECMAScript (Esempio 4)

```js
// my-module.js
const add = (a, b) => a + b;
const sub = (a, b) => a - b;

export default (a, b) => { return a * b; };
export { add, sub };
```

```js
// index-namespace-import.js
import * as myModule from './my-module.js';

console.log(myModule.add(1, 2)); // 3
console.log(myModule.sub(1, 2)); // -1
console.log(myModule.default(1, 2)); // 2
```

namespace import

```js
// index-named-import.js
import multiply, { add, sub } from './my-module.js';

console.log(add(1, 2)); // 3
console.log(sub(1, 2)); // -1
console.log(multiply(1, 2)); // 2
```

named import

CommonJS vs ECMAScript

	Compatibilità con Node.js	Compatibilità con browser	Import dinamico	Import asincrono
CommonJs	✅	✅	✅	
ECMAScript	⚠️ (da v12+)			✅

```
if (condition) {
    const moduleA = require('./moduleA');
}
```

```
import('./moduleA').then(moduleA => {
    moduleA.doSomething();
});
```

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/

Esempio 5

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

Esempio 5

- Creare una cartella
- Posizionarsi con il terminale nella cartella ed eseguire il comando `npm init`
  - Verrà creato il file `package.json` con le dipendenze del progetto.
- Aggiungere express al progetto: `npm install express`
- Cambiamenti
  - `package.json`: nell'elenco delle dipendenze è stato aggiunto express
  - `package-lock.json`: file contenente l'albero delle dipendenze
  - `node_modules`: cartella che contiene il codice dei moduli importanti

Esempio 5

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

Esempio 5

- Creare l'applicazione express

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

- Specificare la gestione della rotta /

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

Esempio 5

- Infine mettere il server in ascolto sulla porta 3000
- ```
  app.listen(3000, ()=> {
     console.log(‘Server is running on http://localhost:3000’);
  });
  ```
  ```

Esempio 5

- Cosa succede visitando la rotta /asw ?

```
Cannot GET /asw
```

Esempio 5

Express utilizza una logica di exact-match, in ordine di definizione delle rotte.

```
app.get('/*', (req, res) => {
    res.setHeader('Content-Type', 'text/plain');
    res.status(404);
    res.send('Page not found');
});
```

Esempio 5.1: Restituire json

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

Esempio 5.1: 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)=>{});
```

Esempio 5.1: 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);
```

Esempio 5.2: Restituire un file html

- Restituire il file `contacts.html` alla rotta /contacts

Esempio 5.2: Restituire un file html

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

```
const path = require('path');
app.get('/contacts', (req, res) => {
    res.sendFile(path.join(__dirname,'contacts.html'));
});
```

Esempio 5.2 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?

Esempio 5.2: 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'));
```

Esempio 5.2 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

Esempio 5.3: 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

Esempio 5.3: Leggere parametri

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

Esempio 5.3: Leggere parametri

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

Template

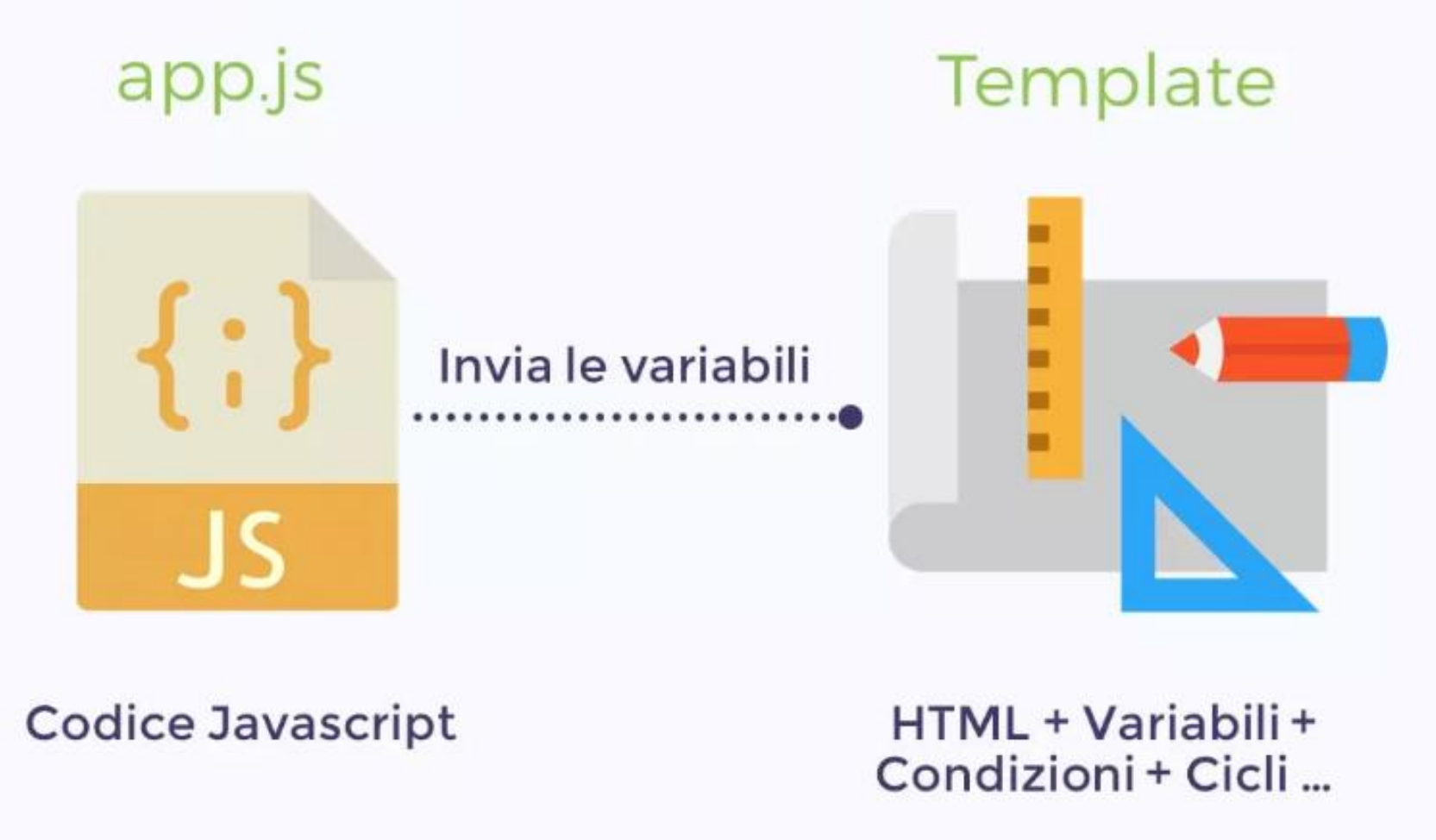
- **Problema**: logica e presentazione non sono separate

```
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

Esempio 6: 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”.

Esempio 6

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

Esempio 6

- 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});
});
```

Esempio 6 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.

Esempio 6 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 2

- 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 2

- Modificare il template:

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

Esercizio 2

- 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/