

Express

Overview

- Minimal node js framework
- Features
 - complex routing
 - req/resp handling
 - middleware
 - server side rendering
- Rapid app development
 - MVC Architecture

basic app

```

1  const express = require('express');
2
3  const app = express();
4
5  app.get('/', (req, res) => {
6    console.log(`Request received`);
7    res.status(200).send('Hello from the server');
8  });
9
10 const port = 3000;
11 app.listen(port, () => {
12   console.log(`App running on port ${port}`);
13 });

```

send file

```
const app = express();

app.get('/', (req, res) => {
  res.status(200).sendFile(path.join(__dirname, '/index.html'));
});

const port = 3000;

app.listen(port, () => {
  console.log(`App running on port ${port}`);
});
```

Rest Client

Visual Studio | Marketplace

Visual Studio Code > Programming Languages > REST Client




REST Client

Huachao Mao |  5,893,395 installs | ★★★★★ (371) | Free

REST Client for Visual Studio Code

[Install](#)

[Trouble Installing?](#) 

Cos'è il routing

- Il **routing** è il modo in cui un'applicazione risponde alle richieste HTTP su determinati percorsi (URL) con metodi specifici (GET, POST, ecc.).

Routing con stringhe

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

app.get('/', (req, res) => {
  res.send('Homepage');
});

app.get('/about', (req, res) => {
  res.send('Pagina About');
});

app.get('/contact', (req, res) => {
  res.send('Contattaci');
});

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

- **app.get(path, callback)** → risponde a una richiesta GET sull'URL path
- **req** e **res** sono gli oggetti della richiesta e della risposta
- Le stringhe come '/', '/about' sono chiamate **percorsi static**
- Le rotte sono **case-sensitive** e **order-sensitive**
 - /about ≠ /About
- Le rotte sono lette nell'ordine in cui sono scritte

Routing sui metodi

```
// GET method route
app.get('/', (req, res) => {
  res.send('GET request to the homepage')
})

// POST method route
app.post('/', (req, res) => {
  res.send('POST request to the homepage')
})
```


Routing sui metodi

```
// GET method route
app.get('/', (req, res) => {
  res.send('GET request to the homepage')
})

// POST method route
app.post('/', (req, res) => {
  res.send('POST request to the homepage')
})
```

checkout(), copy(), **delete()**, **get()**, head(), lock(), merge(), mkactivity(), mkcol(), move(),
m-search(), notify(), options(), patch(), **post()**, purge(), **put()**, report(), search(),
subscribe(), trace(), unlock(), unsubscribe().

Routing sui metodi

```
// GET method route
app.get('/', (req, res) => {
  res.send('GET request to the homepage')
})
```

```
// POST method route
app.post('/', (req, res) => {
  res.send('POST request to the homepage')
})
```

checkout(), copy(), **delete()**, **get()**, head(), lock(), merge(), mkactivity(), mkcol(), move(),
m-search(), notify(), options(), patch(), **post()**, purge(), **put()**, report(), search(),
subscribe(), trace(), unlock(), unsubscribe().

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

Route parameters

- I **route parameters** (o **parametri dinamici**) sono segmenti della URL che fungono da **segnaposto** per valori variabili. Si usano per creare rotte **dinamiche** in Express.

Route parameters

- I **route parameters** (o **parametri dinamici**) sono segmenti della URL che fungono da **segnaposto** per valori variabili. Si usano per creare rotte **dinamiche** in Express.

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

app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params);
});

app.listen(3000, () => {
  console.log('Server avviato su http://localhost:3000');
});
```

Route parameters

- I **route parameters** (o **parametri dinamici**) sono segmenti della URL che fungono da **segnaposto** per valori variabili. Si usano per creare rotte **dinamiche** in Express.

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

app.get('/:userId/books/:bookId', (req, res) => {
  res.send(req.params);
});

app.listen(3000, () => {
  console.log('Server avviato su http://localhost:3000');
});
```

Route parameters

- I **route parameters** (o **parametri dinamici**) sono segmenti della URL che fungono da **segnaposto** per valori variabili. Si usano per creare rotte **dinamiche** in Express.

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

app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params);
});

app.listen(3000, () => {
  console.log('Server avviato su http://localhost:3000');
});
```

<http://localhost:3000/users/34/books/8989>

Route parameters

- I **route parameters** (o **parametri dinamici**) sono segmenti della URL che fungono da **segnaposto** per valori variabili. Si usano per creare rotte **dinamiche** in Express.

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

app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params);
});

app.listen(3000, () => {
  console.log('Server avviato su http://localhost:3000');
});
```

http://localhost:3000/users/34/books/8989



```
{
  "userId": "34",
  "bookId": "8989"
}
```

Static Files

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

Per gestire i file statici, quali immagini, file CSS e file JavaScript, utilizzare la funzione middleware integrata `express.static` in Express.

Fornire il nome della directory che contiene gli asset statici alla funzione middleware `express.static` per iniziare a gestire i file direttamente. Ad esempio, utilizzare il seguente codice per gestire le immagini, i file CSS e i file JavaScript nella directory denominata `public`:

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

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


json response

```
app.get('/', (req, res) => {  
  console.log(`Request received`);  
  res.status(200).json({ message: 'Hello from the server' });  
});
```

Simple post

```
app.post('/', (req, res) => {  
  console.log(`Request received`);  
  res.status(404).json({ message: 'Post Endpont!!!' });  
});
```

Architettura REST

- **REST** (*RE*presentational State *T*ransfer)
- insieme di linee guida o principi per la realizzazione di una architettura di sistema
 - uno stile architetturale
- non si riferisce ad un sistema concreto
- non si tratta di uno standard

Architettura rest

- Client-Server

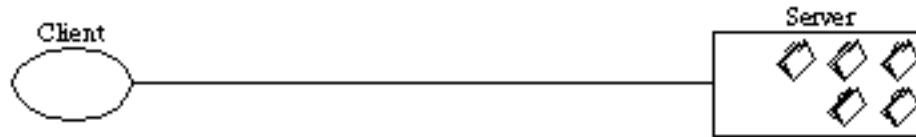


Figure 5-2. Client-Server

- Stateless

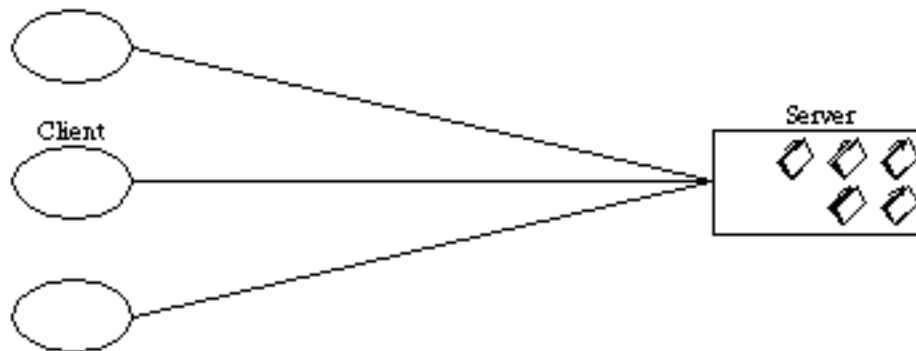


Figure 5-3. Client-Server

Architettura rest

- Cache

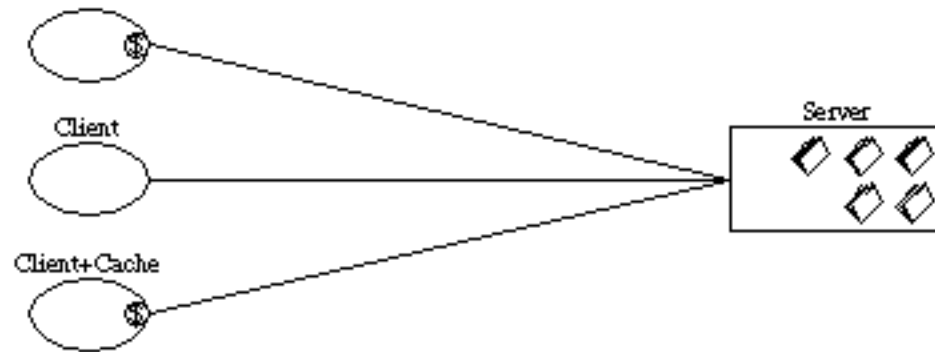


Figure 5-4. Client-Cache-Stateless-Server

- Uniform Interface

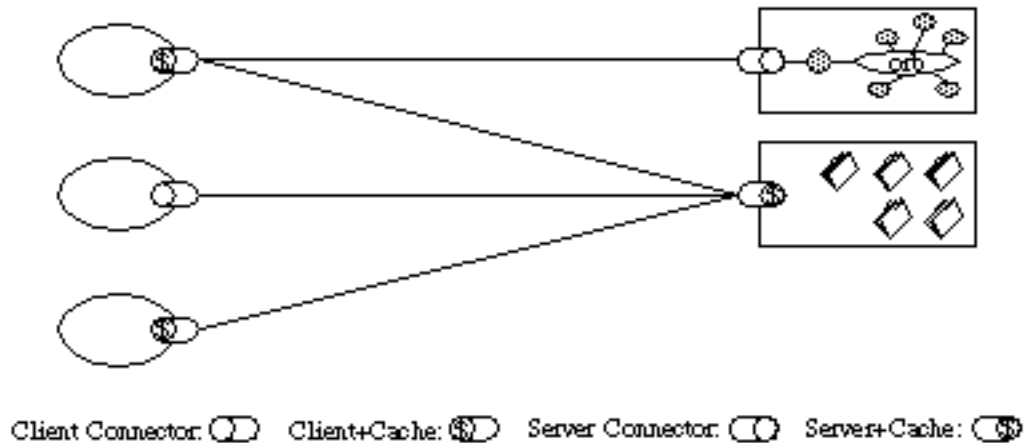


Figure 5-6. Uniform-Client-Cache-Stateless-Server

Che cos'è REST?

- Linee guida per la realizzazione di un servizio web.
- Principi base
 - Identificazione delle risorse
 - Utilizzo esplicito dei metodi HTTP
 - Risorse autodescrittive
 - Collegamenti tra risorse
 - Comunicazione senza stato

Identificazione delle risorse

- Risorsa: qualsiasi elemento oggetto di elaborazione
- Es. un cliente, un libro, un articolo, un qualsiasi oggetto su cui è possibile effettuare operazioni
- Ciascuna risorsa deve essere identificata univocamente
 - Mediante una URI.

URI per risorse

`http://www.myservice.com/clients/1234`

`http://www.myservice.com/orders/2019/98765`

`http://www.myservice.com/products/7654`

`http://www.myservice.com/orders/2019`

`http://www.myservice.com/products?color=red`

- L'interpretazione è desunta dalla semantica delle parole contenute nelle sue parti.
- per un Web Service sono soltanto stringhe
 - *`http://www.myservice.com/prd21/2022ww`*

Operazioni CRUD

- Acronimo per:
create, read (aka retrieve), update, and delete
- Operazioni di base che posso fare su una risorsa
 - **Create** (creare una risorsa)
 - **Read** o **Retrieve** (leggere una risorsa)
 - **Update** (aggiornare una risorsa)
 - **Delete** (eliminare una risorsa)

REST e CRUD

Metodo HTTP	Operazione CRUD	Descrizione
POST	Create	Crea una nuova risorsa
GET	Read	Ottiene una risorsa esistente
PUT	Update	Aggiorna una risorsa o ne modifica lo stato
DELETE	Delete	Elimina una risorsa

Esempio

Risorsa	GET read	POST create	PUT update	DELETE
<i>/books</i>	Ritorna una lista di libri	Crea un nuovo libro	Aggiorna i dati di tutti i libri	Elimina tutti i libri
<i>/books/145</i>	Ritorna uno specifico libro	metodo non consentito (405)	Aggiorna uno specifico libro	Elimina uno specifico libro

REST vs Web

- Nel mondo WEB viene utilizzato il metodo GET per eseguire qualsiasi tipo di interazione con il server.
 - Inserimento:
GET `http://www.myservice.com/addCustomer?name=Rossi`
- Non conforme ai principi REST:
 - GET serve per accedere alla rappresentazione di una risorsa e non per crearne una nuova
- Il body HTTP della POST e della PUT è pensato per il trasferimento della rappresentazione di una risorsa
 - non per eseguire chiamate remote o altre attività simili.

Rappresentazione di Risorse

- Le risorse sono codificate ed inviate al client
 - al suo interno il server le memorizza come vuole
- Caratteristiche della rappresentazione:
 - Understandability
 - Completeness
 - Linkability
- Formati tipici: JSON e XML

Specificare la rappresentazione

- Client e server possono specificare il formato per la risorsa
 - dicono il MIME Type
- Client:
 - **Accept**
- Server:
 - **Content-Type**

```
GET /clienti/1234
HTTP/1.1
Host: www.myapp.com
Accept: application/vnd.myapp.cliente+xml
```

Entry Point

- Le API REST devono specificare uno ed un solo "Entry Point", punto di ingresso
 - La URL base!!
- Informazioni che fornisce l'entry point
 - Information on API version, supported features, etc.
 - A list of top-level collections.
 - A list of singleton resources.
 - Any other information that the API designer deemed useful, for example a small summary of operating status, statistics, etc.

Struttura delle URL

URL	Description
/api	The API entry point
/api/:coll	A top-level collection named "coll"
/api/:coll/:id	The resource "id" inside collection "coll"
/api/:coll/:id/:subcoll	Sub-collection "subcoll" under resource "id"
/api/:coll/:id/:subcoll/:subid	The resource "subid" inside "subcoll"

```

/endpoint
  /collection1
    /resource1
    /resource2
    /resource3
  /collection2
    /resource1
    /resource2

```

...

URL con Varianti

URL	Description
/api/:coll/:id;saved	Identifies the saved variant of a resource.
/api/:coll/:id;current	Identifies the current variant of a resource.

REST e Status Code

- 200 – OK – Tutto bene
- 201 – OK – E' stata creata una nuova risorsa
- 204 – OK – La risorsa è stata cancellata con successo
- 304 – Not modified – I dati non sono cambiati. Il cliente può utilizzare i dati nella cache
- 400 – Bad Request – Richiesta non valida. L'errore esatto dovrebbe essere spiegato nel payload dell' errore (di cui ne parleremo a breve). Per esempio. "Il JSON non è valido"
- 401 – Unauthorized – La richiesta richiede una autenticazione dell'utente
- 403 – Forbidden – Il server ha capito la richiesta, ma in base ai diritti del richiedente l'accesso non è consentito.
- 404 – Not Found – Non vi è alcuna risorsa dietro l'URI richiesto.
- 422 – Unprocessable Entity – deve essere usato se il server non può elaborare il entity, ad esempio se un'immagine non può essere formattata o campi obbligatori sono mancanti nel payload.
- 500 – Internal Server Error – gli sviluppatori di API dovrebbero evitare questo errore. Se si verifica un errore globale dell'applicazione, lo stacktrace deve essere loggato e non inviato nella risposta all'utente.

CRUD Operations

products

users

orders

`http://my-url/addNewProduct`

`/getProduct`

`/updateProduct`

`/deleteProduct`

`/getProductbyOrder`

`/getOrderByUser`

REST Operations

/addNewProduct

/getProduct

/updateProduct

/deleteProduct

/getProductbyOrder

/getOrderbyUser

POST /products

GET /products/3

PUT /products/3

PATCH /products/3

DELETE /products/3

GET /orders/4/products

GET /users/9/orders

JSON formatting

```
{  
  "id": 1,  
  "name": "cerulean",  
  "year": 2000,  
  "color": "#98B2D1",  
  "pantone_value": "15-4020"  
}
```

JSON formatting

JSEND

<https://github.com/omniti-labs/jsend>

```
{
  "id": 1,
  "name": "cerulean",
  "year": 2000,
  "color": "#98B2D1",
  "pantone_value": "15-4020"
}
```



```
{
  "status": "success",
  "data": {
    "id": 1,
    "name": "cerulean",
    "year": 2000,
    "color": "#98B2D1",
    "pantone_value": "15-4020"
  }
}
```

JSON formatting

JSEND

<https://github.com/omniti-labs/jsend>

```
{
  "id": 1,
  "name": "cerulean",
  "year": 2000,
  "color": "#98B2D1",
  "pantone_value": "15-4020"
}
```



```
{
  "status": "success",
  "data": {
    "id": 1,
    "name": "cerulean",
    "year": 2000,
    "color": "#98B2D1",
    "pantone_value": "15-4020"
  }
}
```

1. [JSON API](#) - JSON API covers creating and updating resources as well, not just responses.
2. [JSend](#) - Simple and probably what you are already doing.
3. [OData JSON Protocol](#) - Very complicated.

Comunicazione Stateless

- **comunicazione stateless:** ciascuna richiesta non ha alcuna relazione con le richieste precedenti e successive
 - La responsabilità della gestione dello stato dell'applicazione non deve essere conferita al server, ma rientra nei compiti del client.
 - La principale ragione di questa scelta è la scalabilità: mantenere lo stato di una sessione ha un costo in termini di risorse sul server e all'aumentare del numero di client tale costo può diventare insostenibile.
 - Inoltre, con una comunicazione senza stato è possibile creare cluster di server che possono rispondere ai client senza vincoli sulla sessione corrente, ottimizzando le prestazioni globali dell'applicazione.

Stateless!!!

- Lo stato va mantenuto nel client
 - Il server per rispondere non deve ricordare una richiesta precedente
- Esempio
 - paging:
 - <https://reqres.in/api/users?page=1>
 - ~~<https://reqres.in/api/users?page=nextpage>~~
 - login
 - ogni richiesta è autenticata singolarmente

REST API

GET

```
app.get('/api/v1/products', (req, res) => {  
  res.status(200).json({  
    status: 'success',  
    data: {  
      products: products,  
    },  
  });  
});
```

GET

```
app.get('/api/v1/products/:id', (req, res) => {
  console.log(req.params);

  const prod = products.find((el) => el.id == req.params.id);
  console.log(prod);
  if (prod == undefined) {
    res.status(404).json({
      status: 'fail',
      message: 'ID non trovato',
    });
  } else {
    res.status(200).json({
      status: 'success',
      data: {
        product: prod,
      },
    });
  }
});
```

POST

```
app.post('/api/v1/products', (req, res) => {
  const newId = products[products.length - 1].id + 1;
  const newProd = Object.assign({ id: newId }, req.body);

  products.push(newProd);
  res.status(201).json({
    status: 'success',
    data: { product: newProd },
  });
});
```

PUT/PATCH

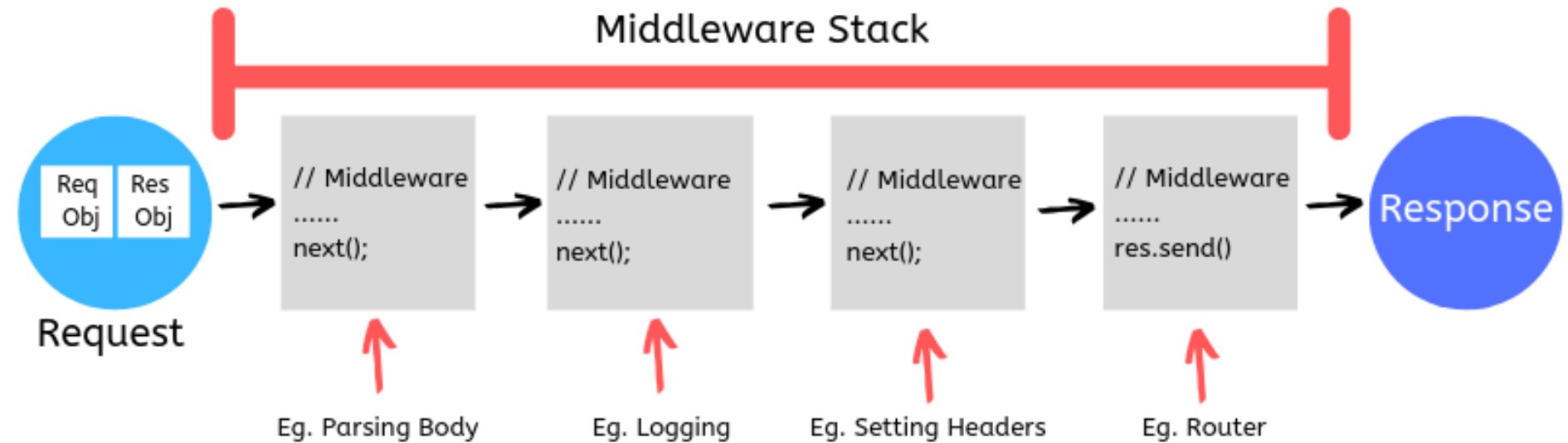
```
app.patch('/api/v1/products/:id', (req, res) => {
  const prod = products.find((el) => el.id == req.params.id);
  if (prod == undefined) {
    res.status(404).json({
      status: 'fail',
      message: 'ID non trovato',
    });
  } else {
    // Update ....

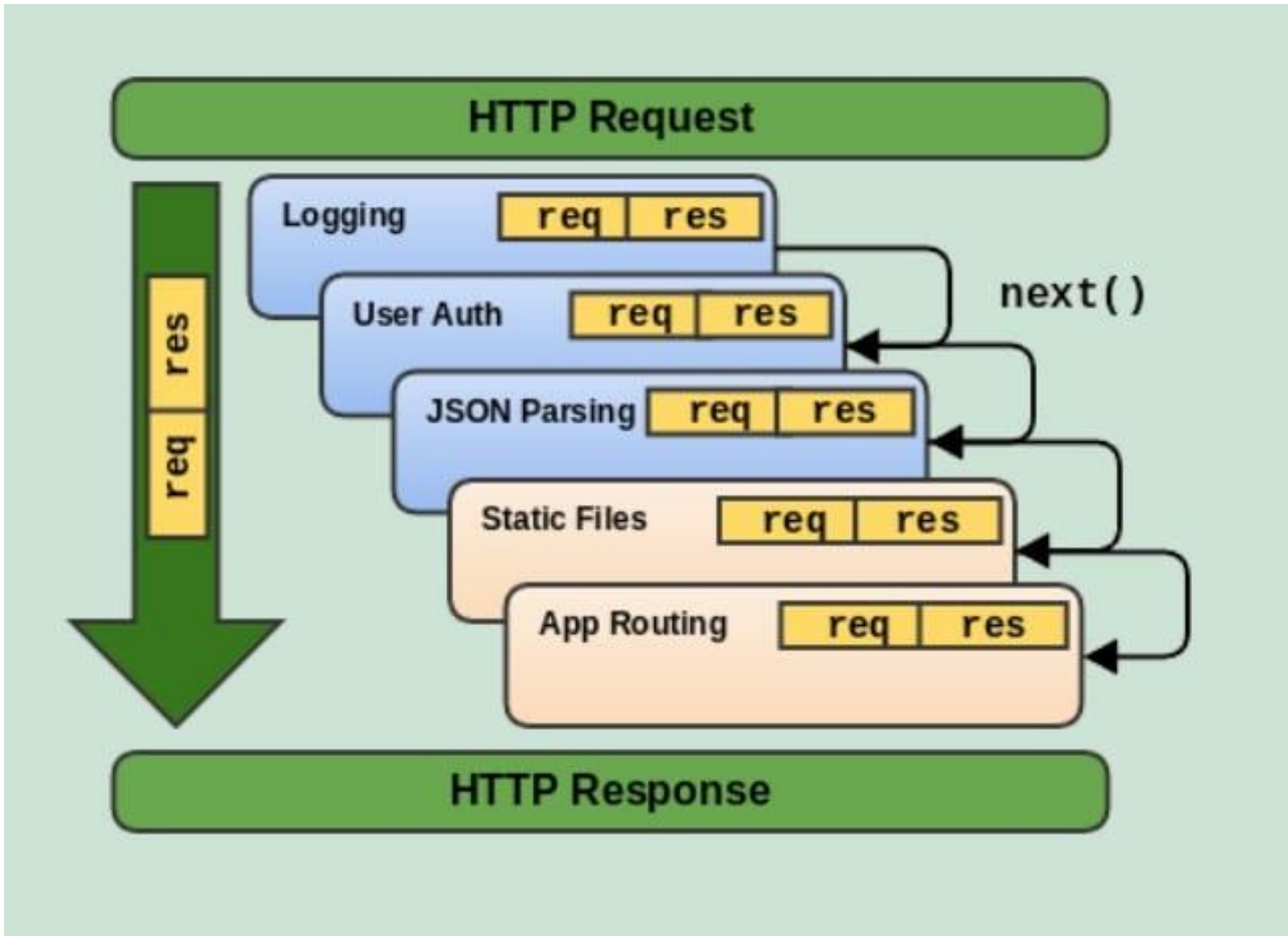
    res.status(200).json({
      status: 'success',
      data: {
        product: prod,
      },
    });
  }
});
```

DELETE

```
app.delete('/api/v1/products/:id', (req, res) => {
  const prod = products.find((el) => el.id == req.params.id);
  if (prod == undefined) {
    res.status(404).json({
      status: 'fail',
      message: 'ID non trovato',
    });
  } else {
    // Delete ....
    res.status(204).json({
      status: 'success',
      data: null,
    });
  }
});
```

Middleware





Custom Middleware

```
app.use(function (req, res, next) {
  console.log('Hello from the middleware !');
  next();
});

app.use('/api', function (req, res, next) {
  console.log('This middleware handles the data route');
  next();
});
```

Third Party middleware

```
const morgan = require('morgan');  
app.use(morgan('dev'));
```

npm install morgan

<https://github.com/expressjs/morgan>

Routing

<https://expressjs.com/en/guide/routing.html>

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes

get(), post(), put()...

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

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

- get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, msearch, notify, subscribe, unsubscribe, patch, search e connect

multiple handlers

```
var cb0 = 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 C!');  
}  
  
app.get('/example/c', [cb0, cb1, cb2]);
```

Un array di funzioni callback possono gestire una route.

Order of routes

```
app.get("/", (req, res) => {
  res.send("Home page");
});

app.get("/page", (req, res) => {
  res.send("A static page");
});

app.get("/:post", (req, res) => {
  res.send("Single post");
});

app.get("*", (req, res) => {
  res.send("Any");
});
```



```
app.get("/", (req, res) => {
  res.send("Home page");
});

app.get("/:post", (req, res) => {
  res.send("Single post");
});

app.get("/page", (req, res) => {
  res.send("A static page");
});

app.get("*", (req, res) => {
  res.send("Any");
});
```

- parametric path inserted just before a literal one takes the precedence over the literal one

route()

- È possibile creare handler di route concatenabili per un percorso di route, utilizzando **app.route()**.

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


Routers

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

// middleware that is specific to this router
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});

// define the home page route
router.get('/', function(req, res) {
  res.send('Birds home page');
});

// define the about route
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

La classe `express.Router` crea handler di route modulari e montabili.

Un'istanza Router è un middleware e un sistema di routing completo; per questa ragione, spesso si definisce "mini-app".

`/birds/`

`/birds/about`

```
var birds = require('./birds');
...
app.use('/birds', birds);
```

param()

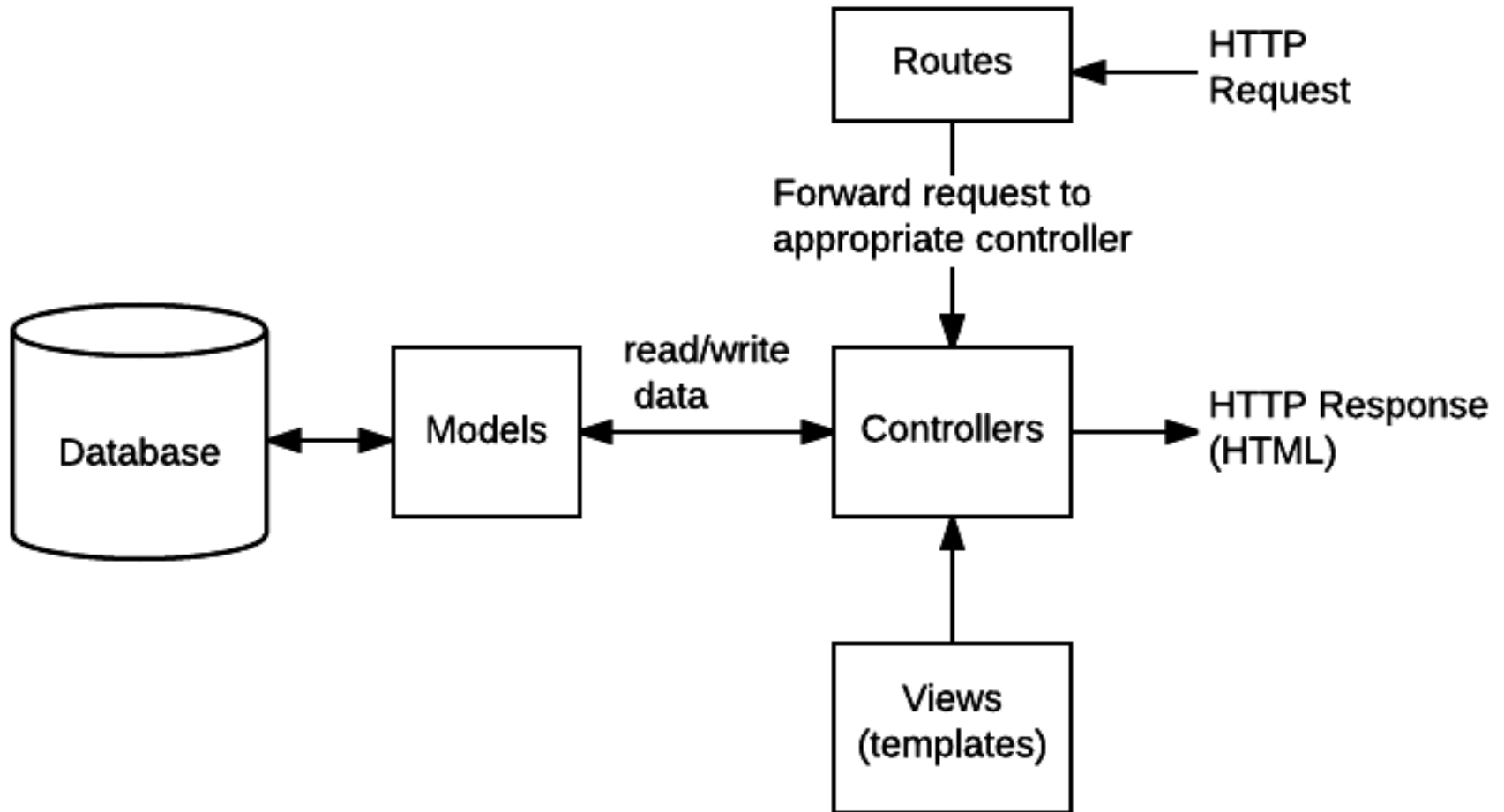
```
const express = require("express");
const router = express.Router();

router.param("userId", (req, res, next, id) => {
  console.log("This function will be called first");
  next();
});

router.get("/user/:userId", (req, res) => {
  console.log("Then this function will be called");
  res.end();
});

// Export router
module.exports = router;
```

MVC Architecture



Environment

- Environment variables allowing apps to behave differently based on the environment
- Externalize all environment specific parameters
- Examples
 - Which HTTP port to listen on
 - What path and folder your files are located in, that you want to serve
 - Pointing to a development, staging, test, or production database

Examples

```
const port = process.env.PORT;

app.listen(port, () => {
  console.log(`App running on port ${port}`);
});
```

- PORT=8626 node server.js
- PORT=8626 NODE_ENV=development node server.js

```
console.log(app.get('env'));

if (app.get('env') == 'development') {
  app.use(morgan('dev'));
}
```

.env file

.env

```
PORT=8765  
NODE_ENV=development  
  
USERNAME=loreti  
PASSWORD=12345
```

server.js

```
const dotenv = require('dotenv');  
dotenv.config();
```

```
const dotenv = require('dotenv');  
dotenv.config({ path: '/custom/path/to/.env' });
```

config module

config.js

```
const dotenv = require('dotenv');
dotenv.config();

module.exports = {
  username: process.env.USER_NAME,
  password: process.env.PASSWORD,
  port: process.env.PORT,
};
```

server.js

```
const { port, username, password } = require('./config');
```