

PROGRAMMAZIONE WEB

prima parte: Node, Angular, React

GIT THE PRINCESS!

HOW TO SAVE THE PRINCESS
USING 8 PROGRAMMING
LANGUAGES

BY  toggly
Goon Squad

YOU HAVE JAVASCRIPT



YOU SPEND HOURS
PICKING LIBRARIES,
SETTING UP NODE &
BUILDING A FRAMEWORK
FOR THE CASTLE.



BY THE TIME
YOU'RE FINISHED WITH
THE FRAMEWORK,
THE FORT HAS
BEEN ABANDONED
AND THE PRINCESS
HAS MOVED TO
ANOTHER CASTLE



YOU HAVE C



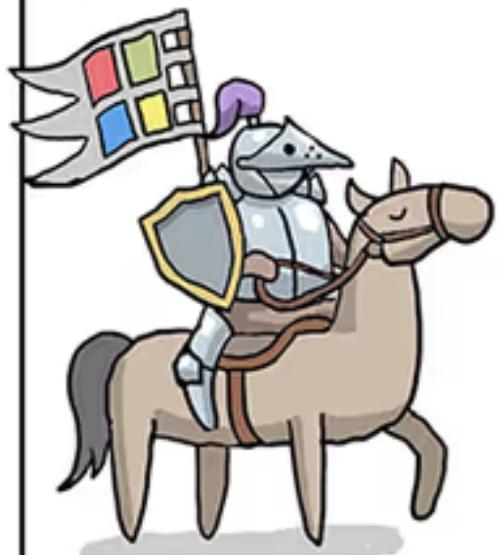
YOU HAVE A LIBRARY
FOR A CASTLE &
A LIBRARY FOR THE
PRINCESS -
CHARGE!



YOU RESCUE THE PRINCESS
HER DOG, HER ENTIRE
WARDROBE & EVERYTHING SHE
HAS EVER EATEN...
**FUCK - DID I FORGET A
NULL-TERMINATOR ?**



YOU HAVE C#



YOU SPEND HOURS
TRYING TO EXPRESS THE
ENTIRE RESCUE IN A
SINGLE LINQ QUERY



YOU GIVE UP AND GO
TO STACKOVERFLOW TO
HAVE JON SKEET
RESCUE THE PRINCESS
FOR YOU.



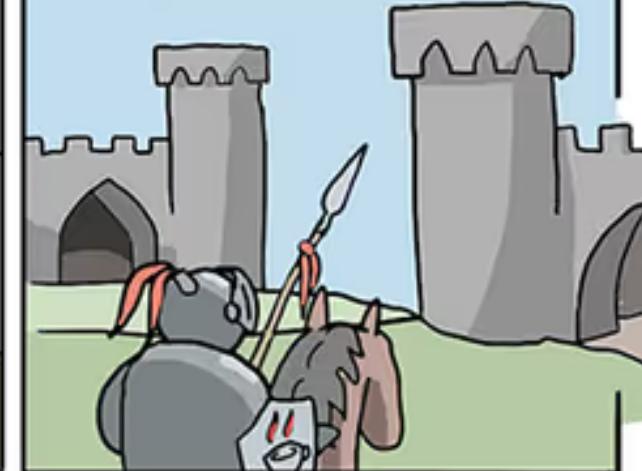
YOU HAVE JAVA



YOU QUICKLY DEPLOY
THE RESCUE
TO PRODUCTION



YOU DISCOVER YOU'VE
LOADED TWO VERSIONS
OF THE CASTLE
BUT NO PRINCESS



YOU HAVE LISP



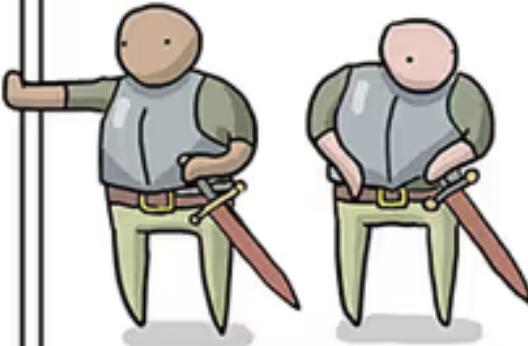
(((((((()))))))))))))))))
((((()))))))))))))))))))))
((((()))))))))))))))))
((((()))))))))))))
((((()))))))))
((((())))))))
((((())))))
((((())))))



YOU HAVE GO



WE DON'T SUPPORT FREEING CAPTURED
PRINCESSES, WE ALREADY HAVE THESE
FREE PRINCESSES IN THE STANDARD LIBR...



...WAIT, IS
THIS THE
PRINCESS
FROM THE
JAVA PANEL?



YOU HAVE PASCAL

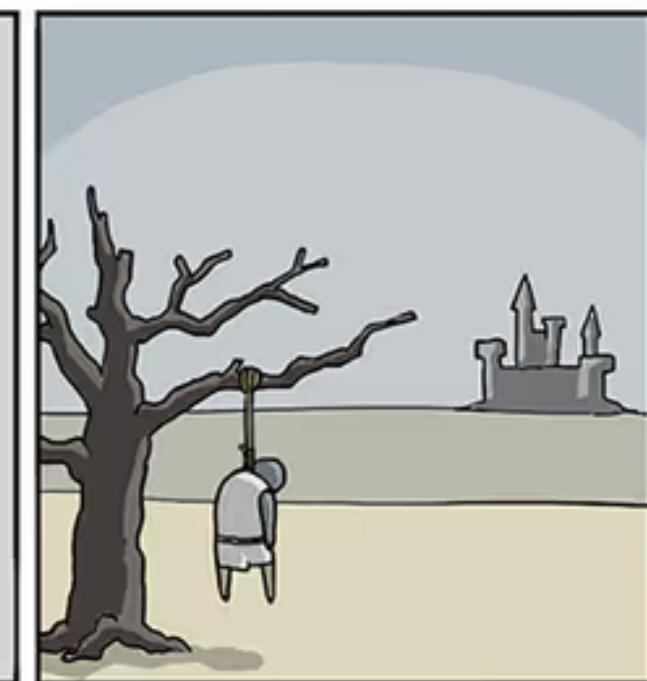


YOU DECLARE
YOUR PRINCESS,
CASTLE &
RESCUE PLAN



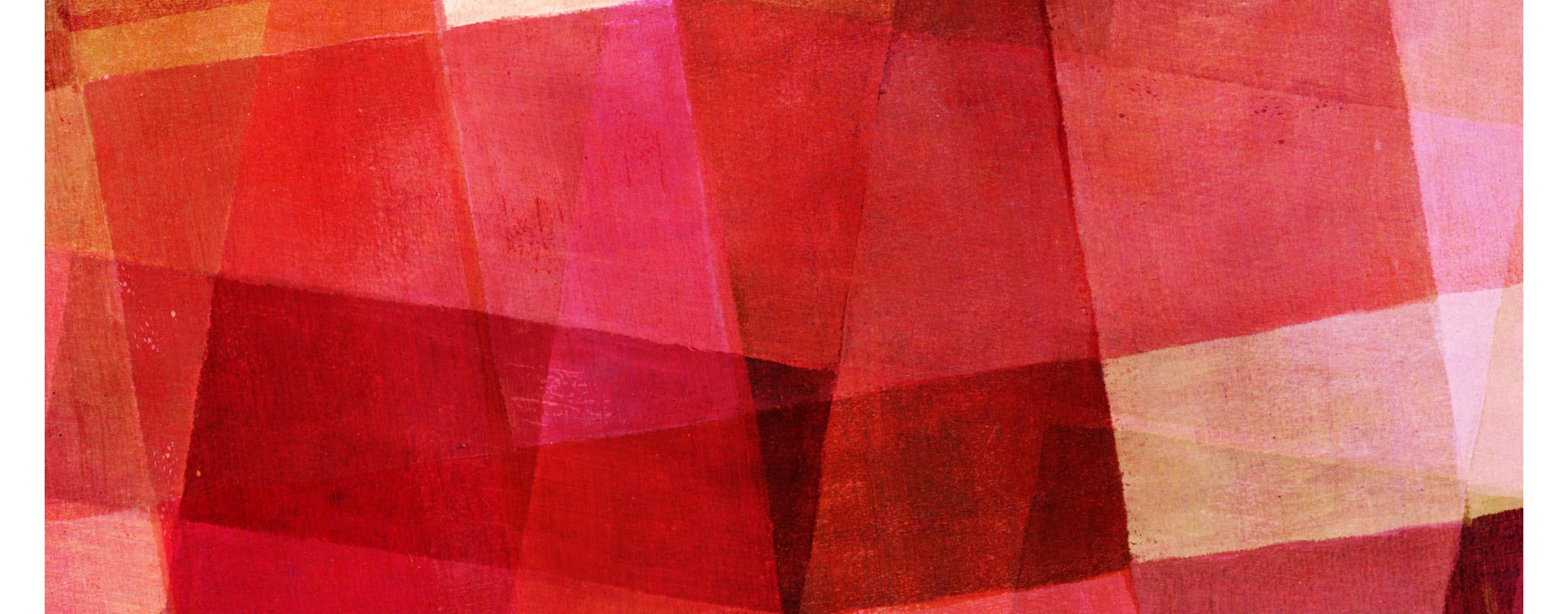
THEN YOU GO FOR
A DRINK & FORGET ABOUT
THE IMPLEMENTATION





MART VIRKUS '16

 toggl

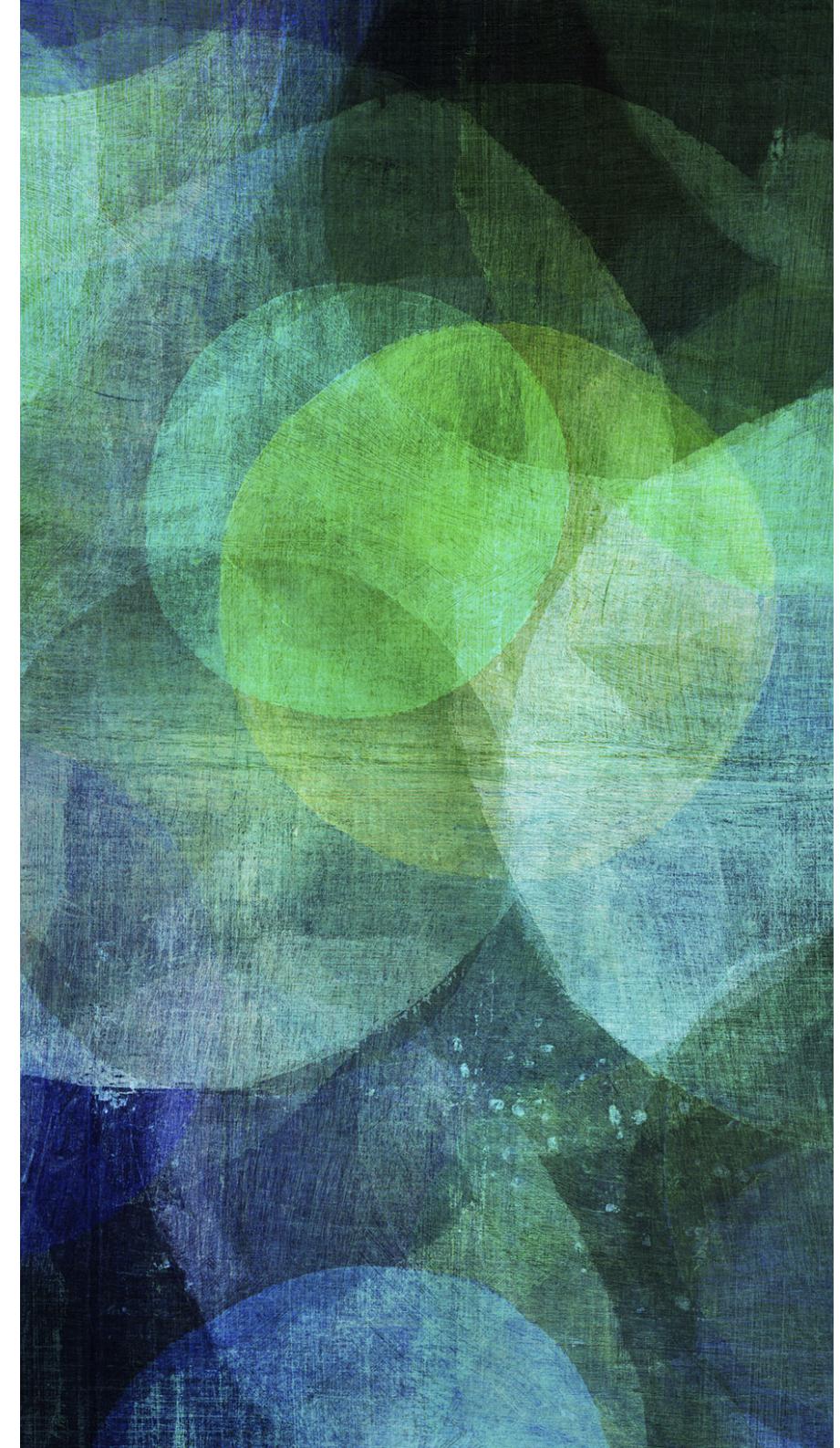


STRUTTURA ARCHITETTURALE

delle moderne applicazioni web

ARCHITETTURA A STRATI

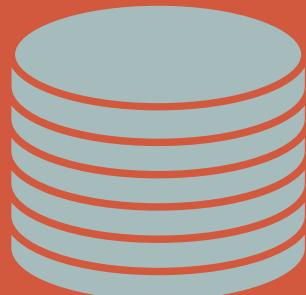
principi di base: separation of concerns, low coupling, high cohesion



ELEMENTI PRINCIPALI

Third Party
Services

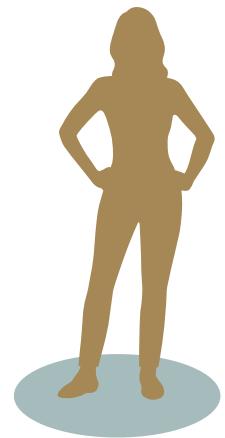
Persistence



Application
Business Logic



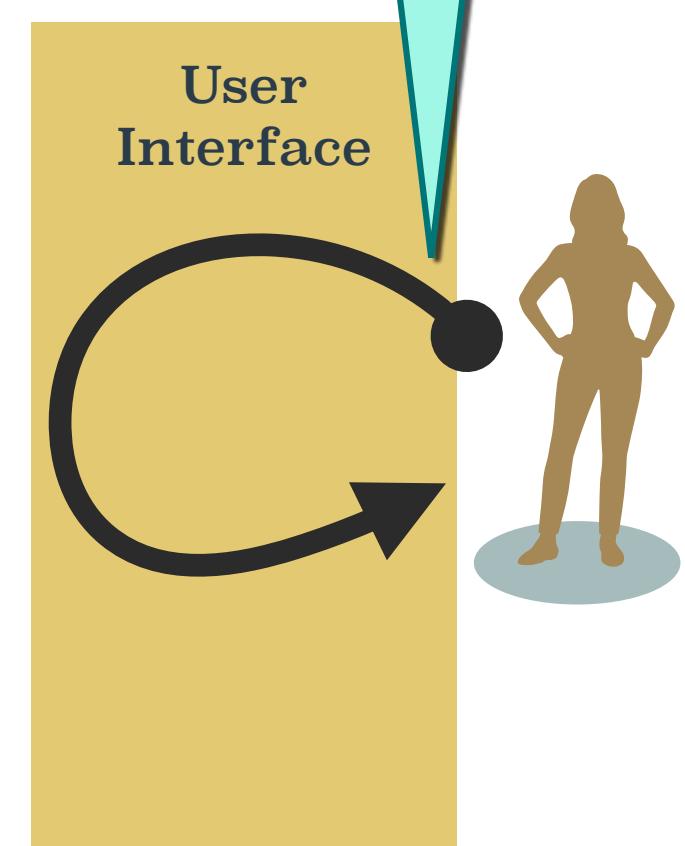
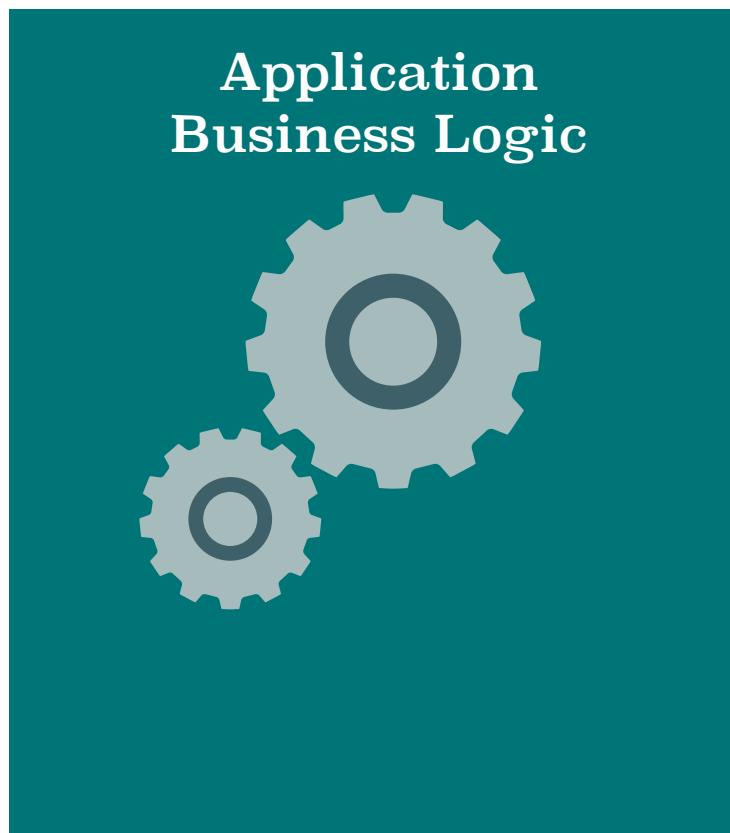
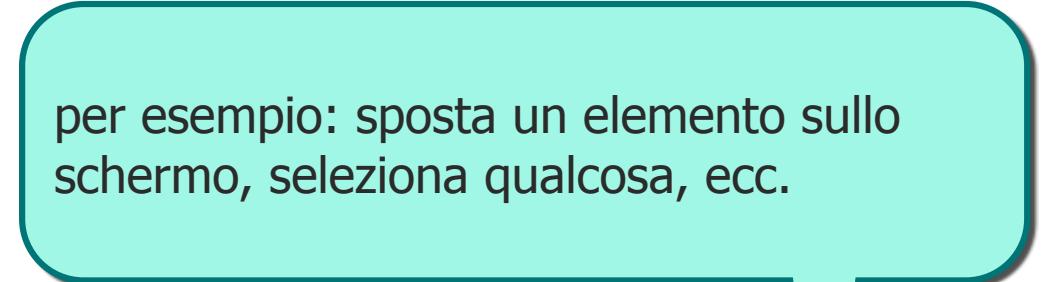
User
Interface



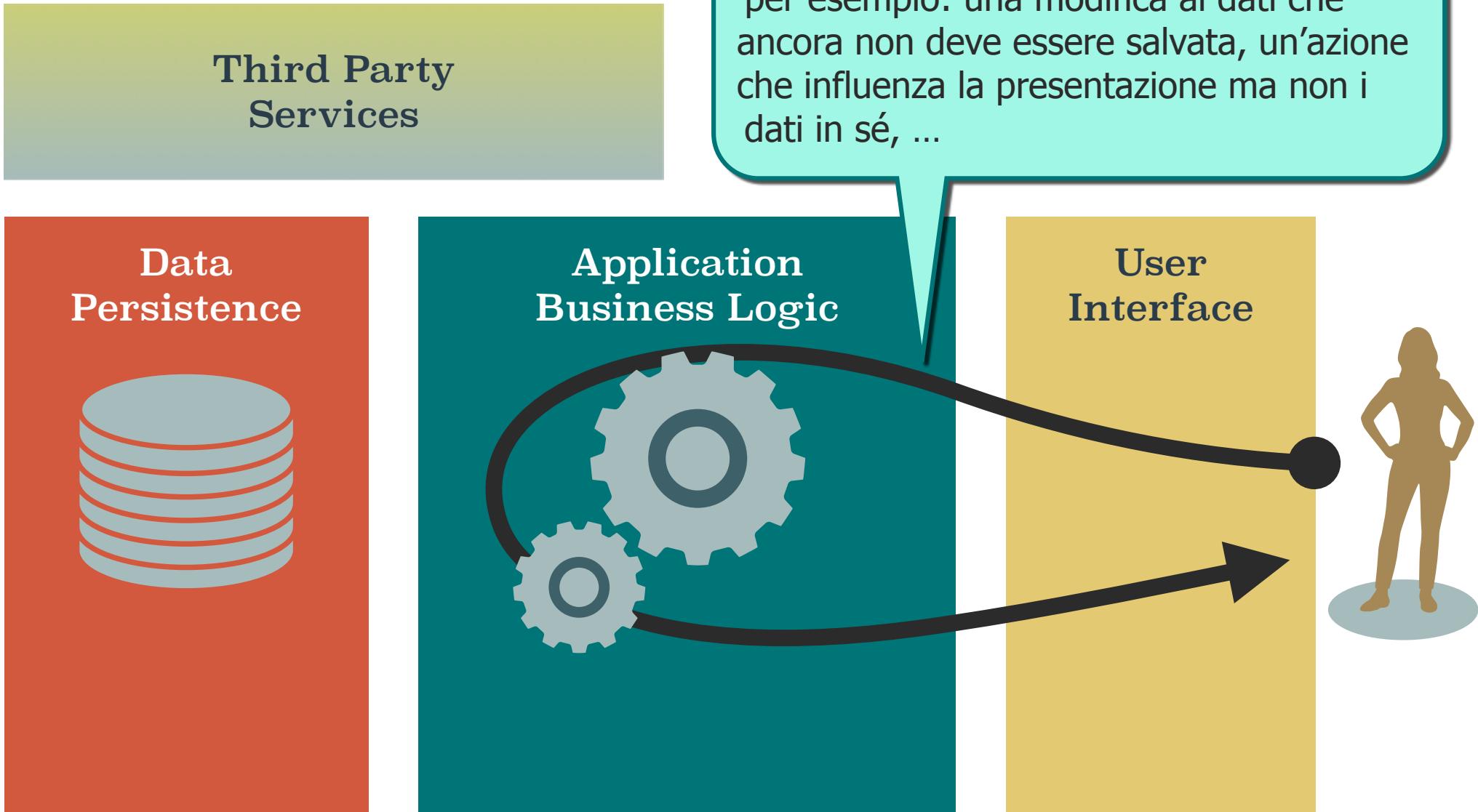
FLOW 1: RETRIEVAL + PRESENTATION



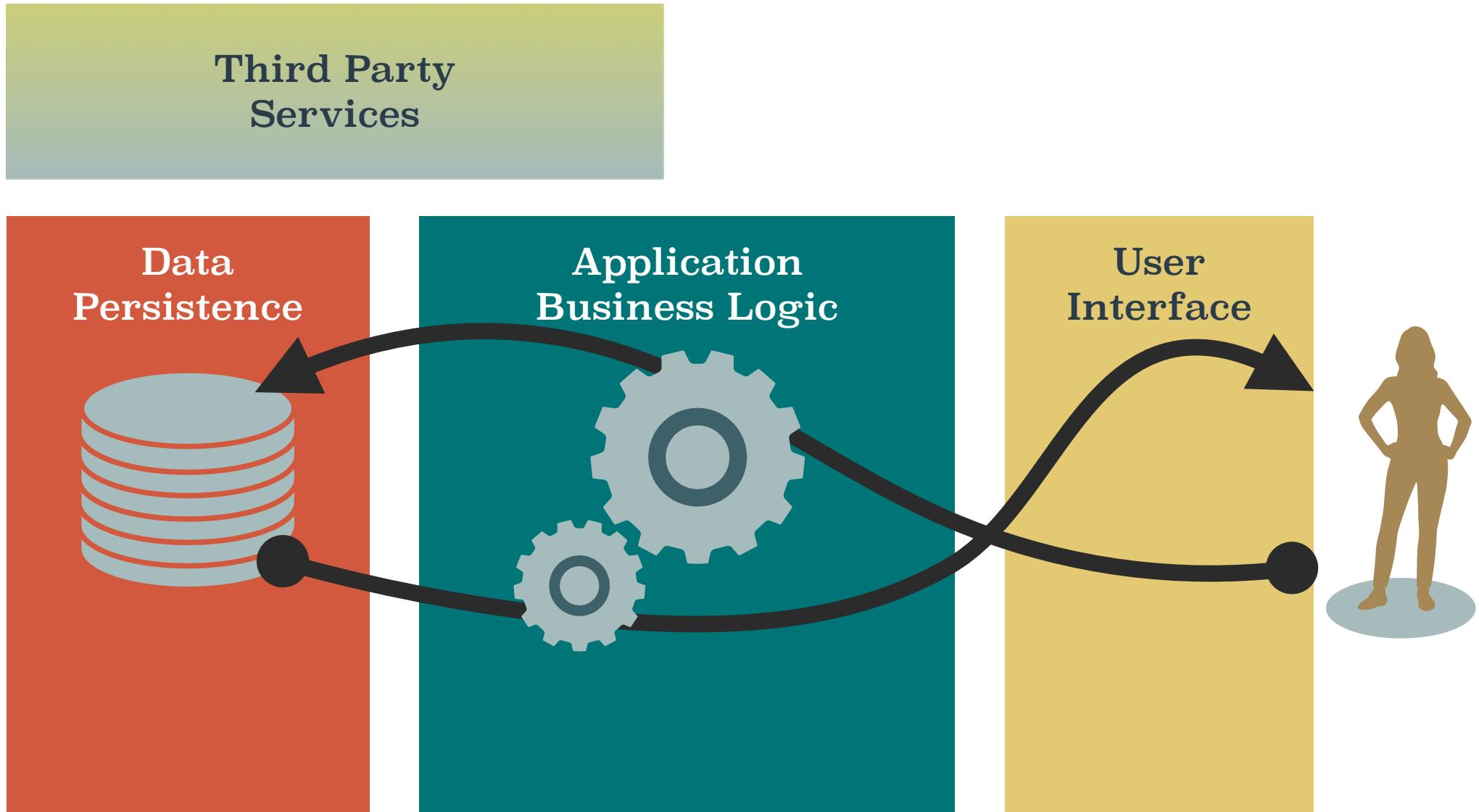
FLOW 2A: USER ACTION, UI-ONLY



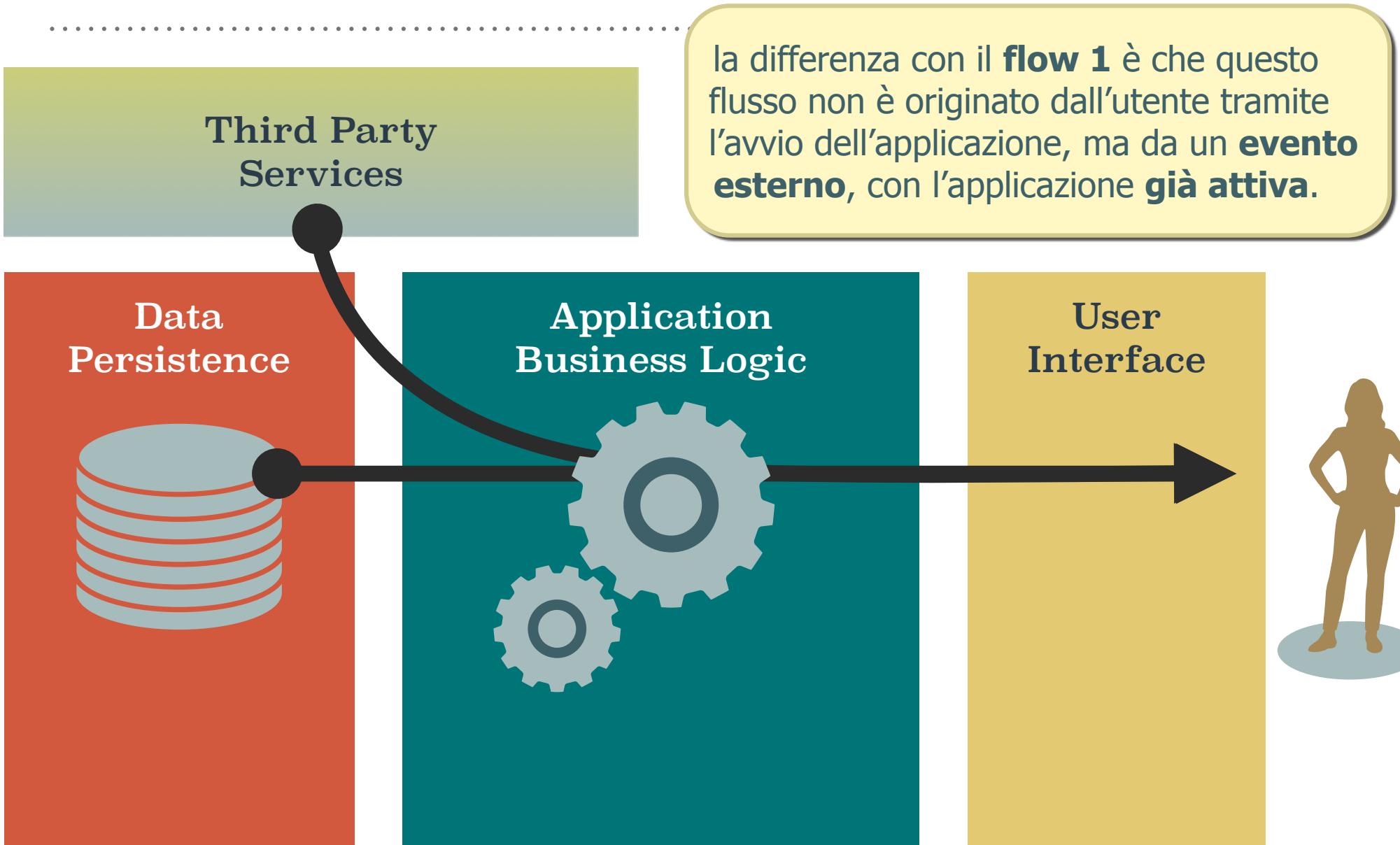
FLOW 2B: USER ACTION, NON-PERSISTENT



FLOW 2C: USER ACTION WITH PERSISTENT CHANGE

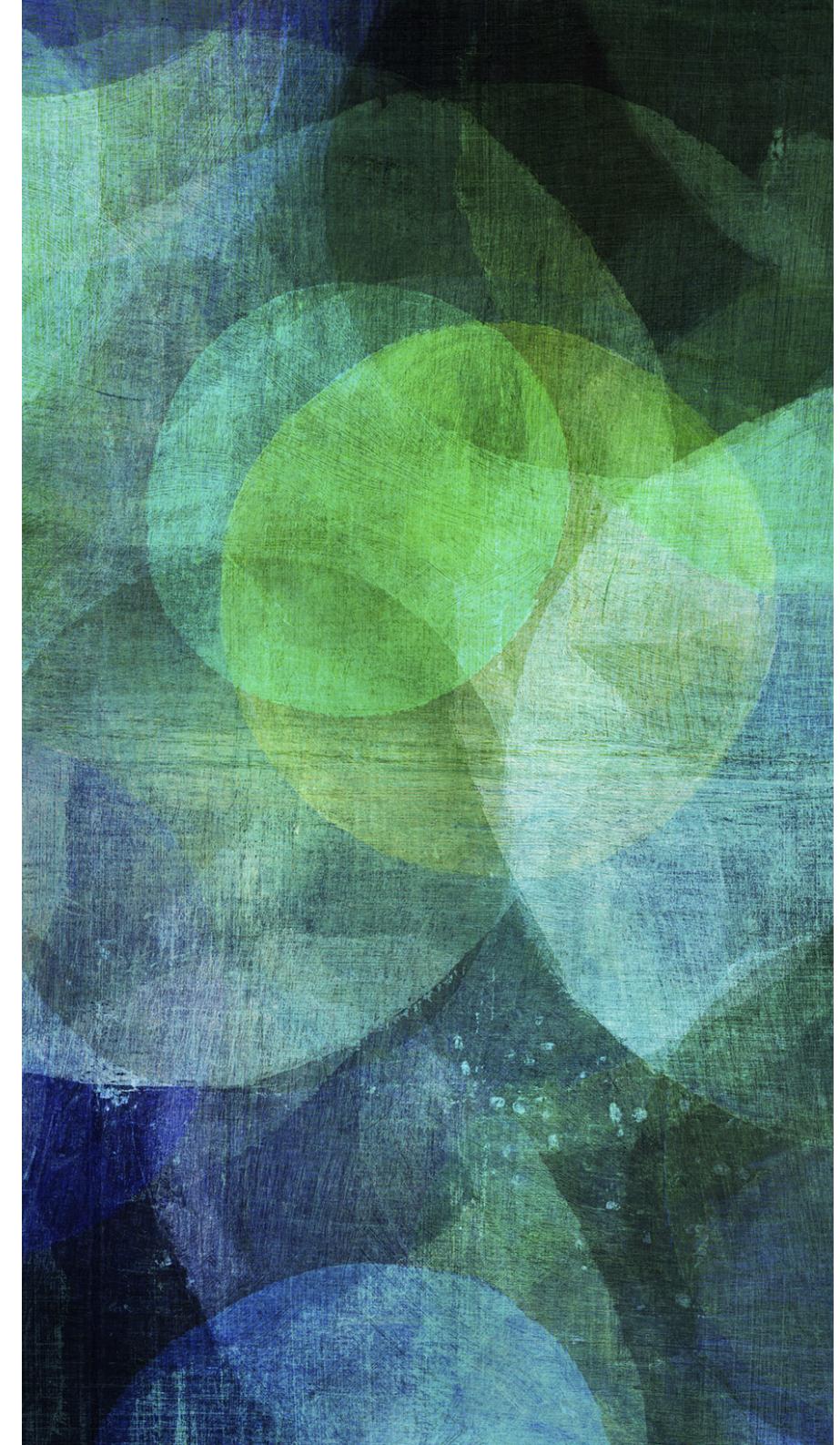


FLOW 3: EXTERNAL UPDATE

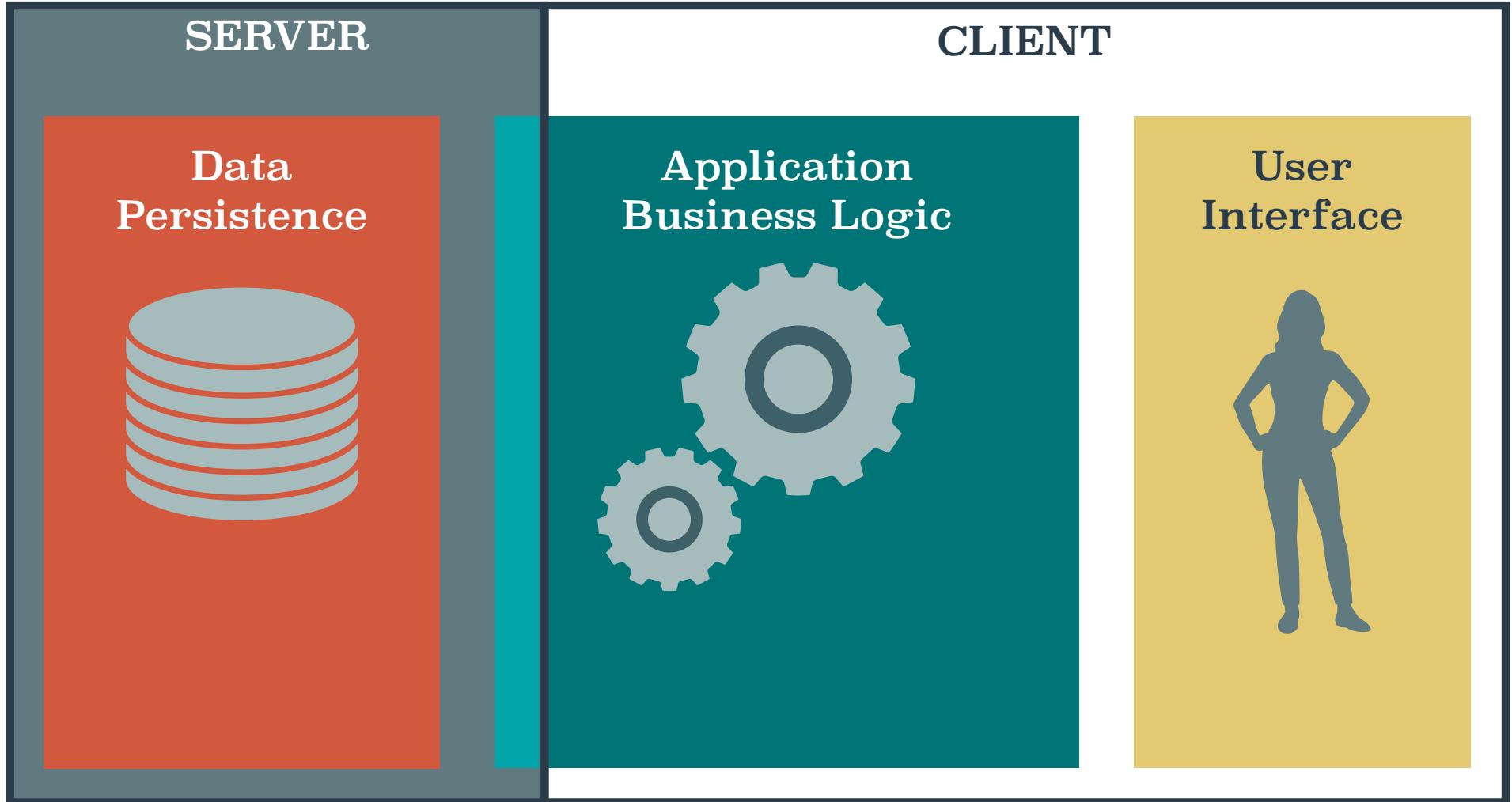


GLI STRATI NELLA ARCHITETTURA CLIENT/SERVER

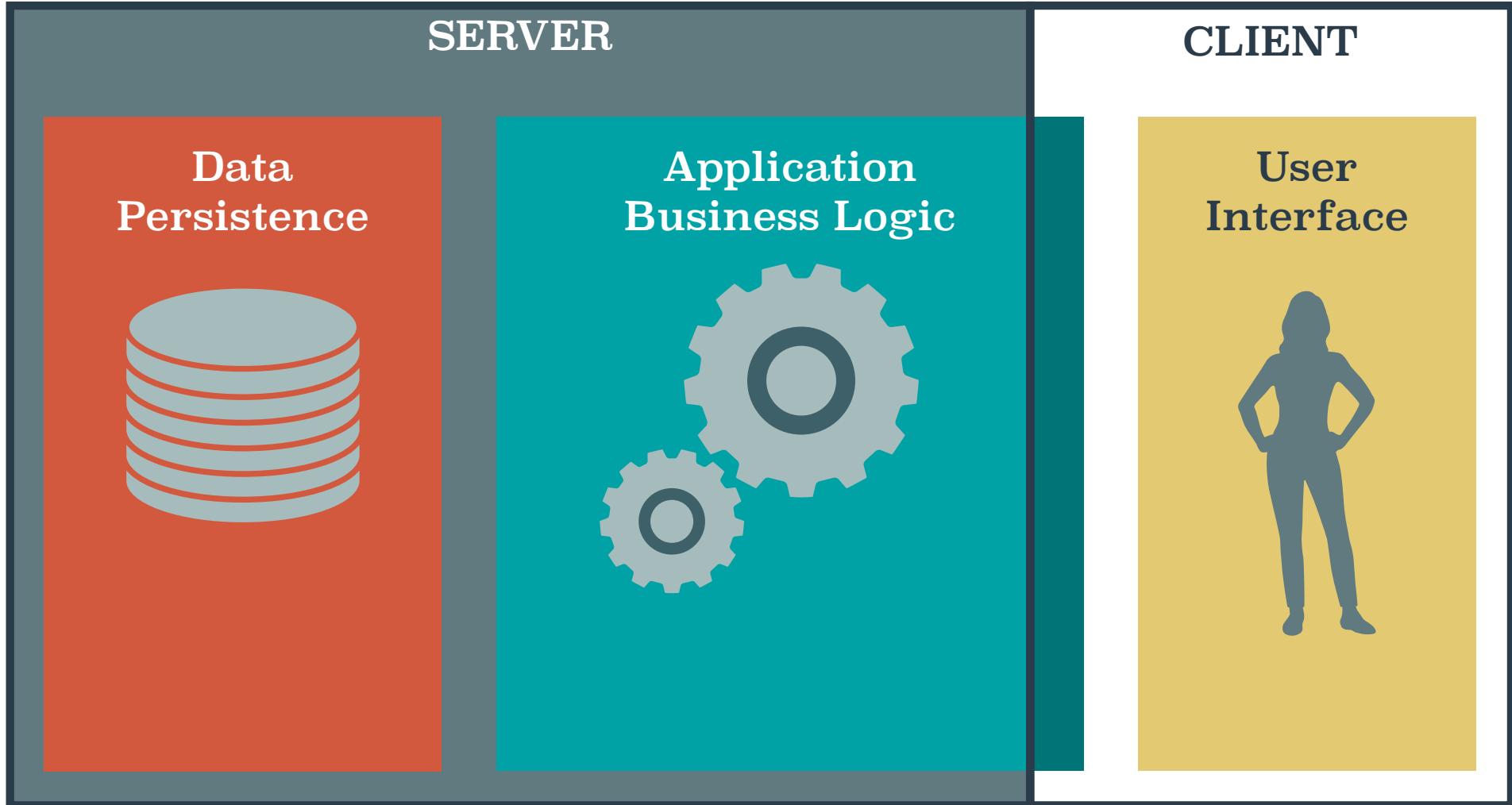
protocollo http



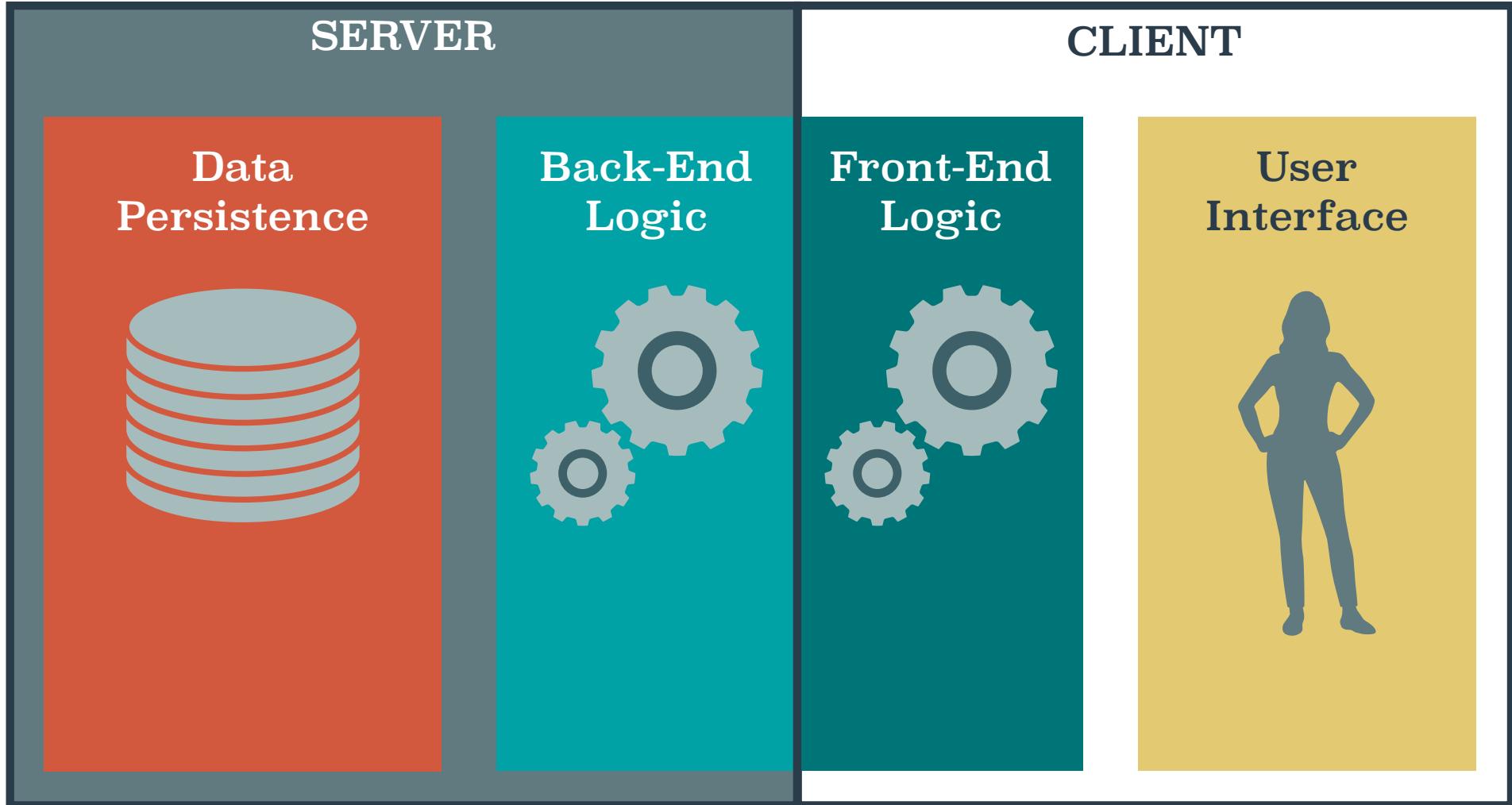
SERVER-SIDE VS CLIENT-SIDE: **CLIENT-SIDE LOGIC**



SERVER-SIDE VS CLIENT-SIDE: SERVER-SIDE LOGIC



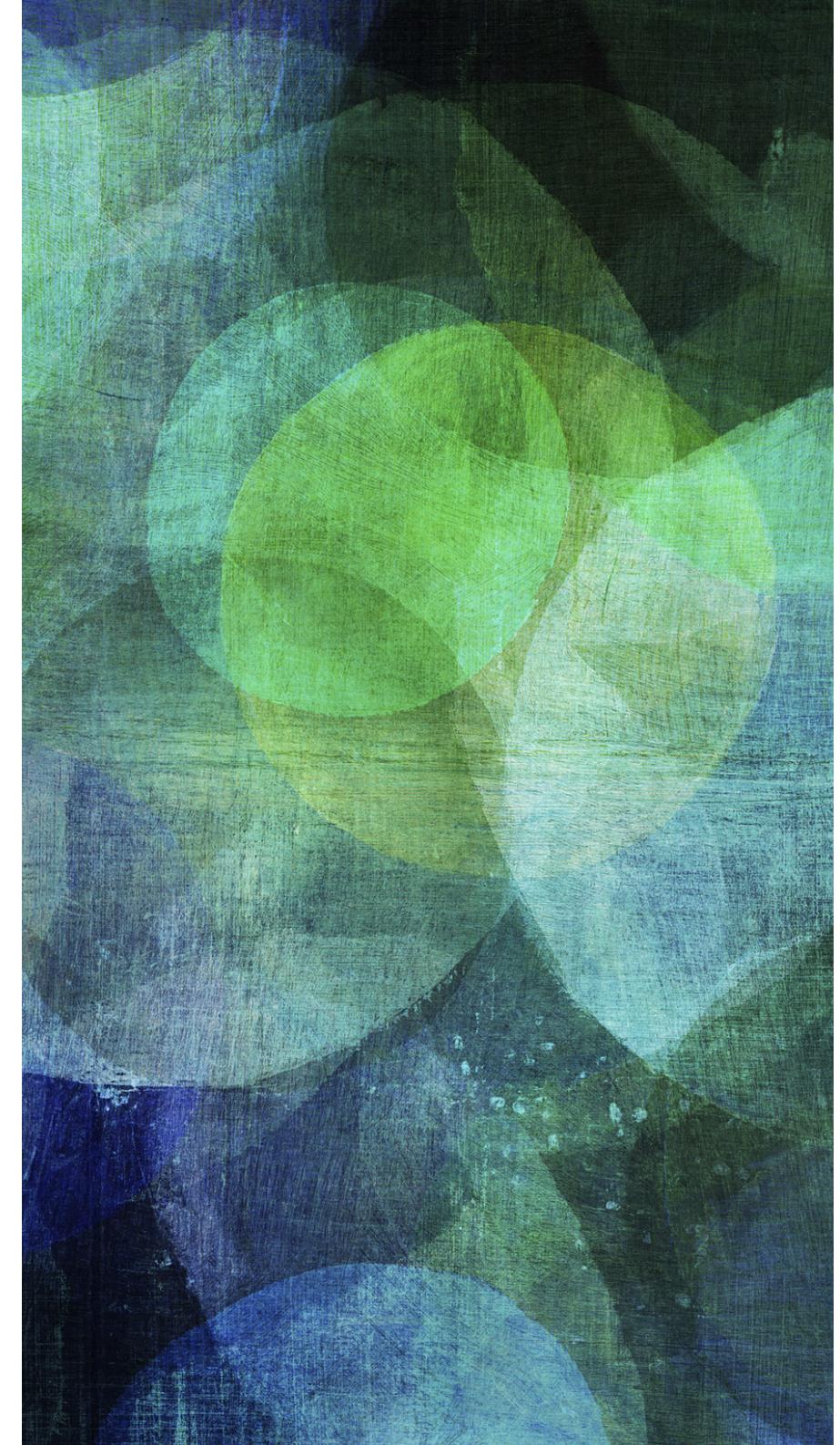
SERVER-SIDE VS CLIENT-SIDE: TWO-SIDED LOGIC



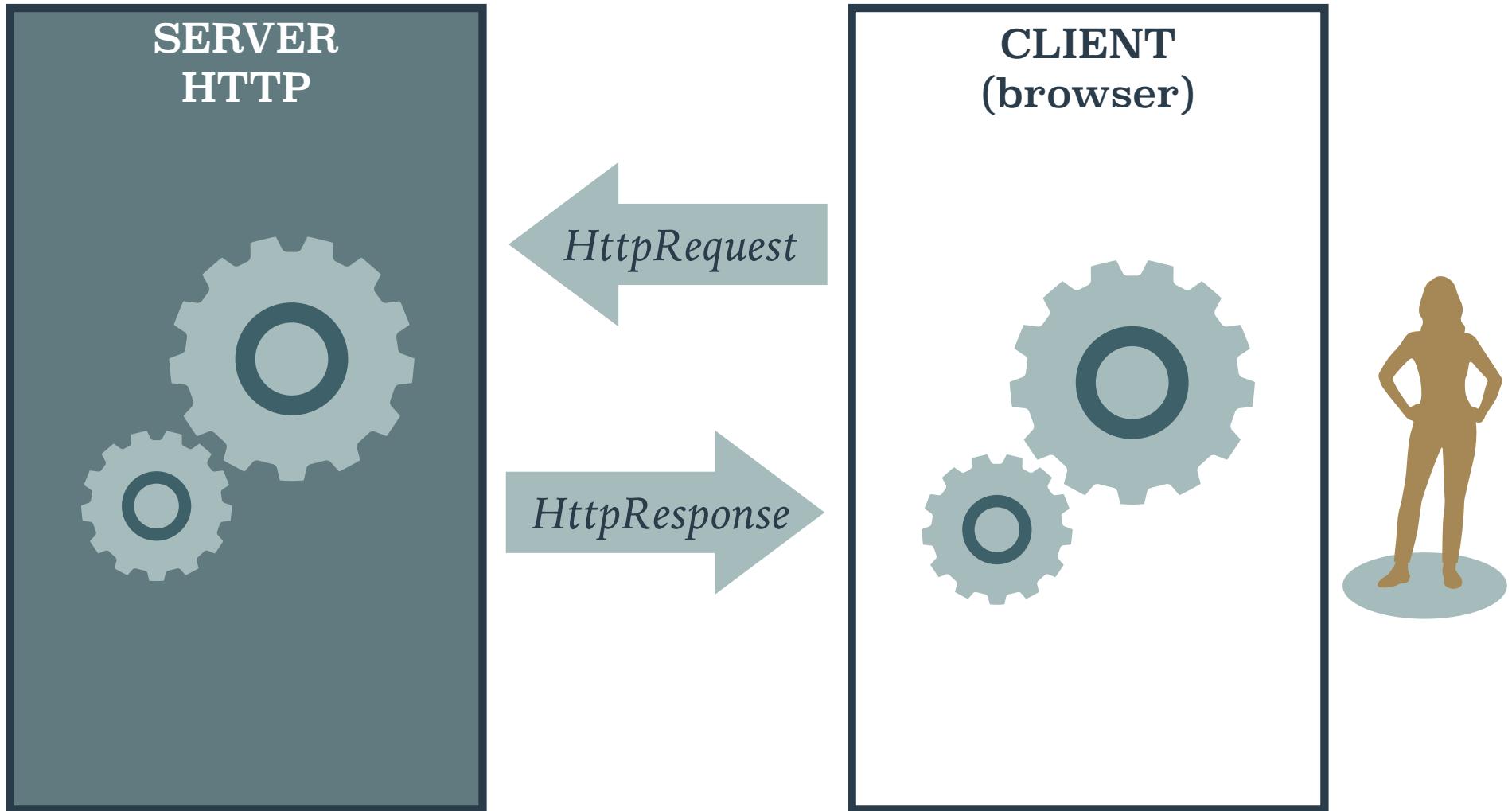
OFFLINE-FIRST APPS



BASARE UN'APP WEB SU HTTP



APPLICAZIONI CLIENT/SERVER BASATE SU HTTP



I CONCETTI FONDANTI DEL WEB

URL = Uniform Resource Locator

- una stringa di testo in grado di identificare qualsiasi **risorsa**

protocol://domain[:port][/path]/resource[?querystring][#fragment]

http://mia.applicazione.org/utenti/profilo/martina.html?opzione=tutti

protocollo client-server **http** = HyperText Transfer Protocol

- un **server** http mette a disposizione **specifiche risorse**
 - le risorse su un dato server sono organizzate in diverse **location**
 - possono essere cartelle in un file system, diversi computer in una sottorete, diverse "posizioni" all'interno di un'applicazione
- un **client** http (**browser**) può richiedere una risorsa tramite una URL
 - il nome del dominio identifica il server a cui porre la richiesta
 - il server è responsabile di interpretare il resto della URL per determinare la corretta risorsa e inviarla in risposta

opzionalmente la URL può indicare delle **opzioni** per meglio specificare la risorsa che sta richiedendo

- le **opzioni** sono precedute da un **?** e sono costituite da **copie attributo=valore** separate dal simbolo **&**
 - http://mia.app.org/gatti.html?nome=felix&tipo=soriano&residenza=grugliasco

SIMPLE SERVER (NODE)

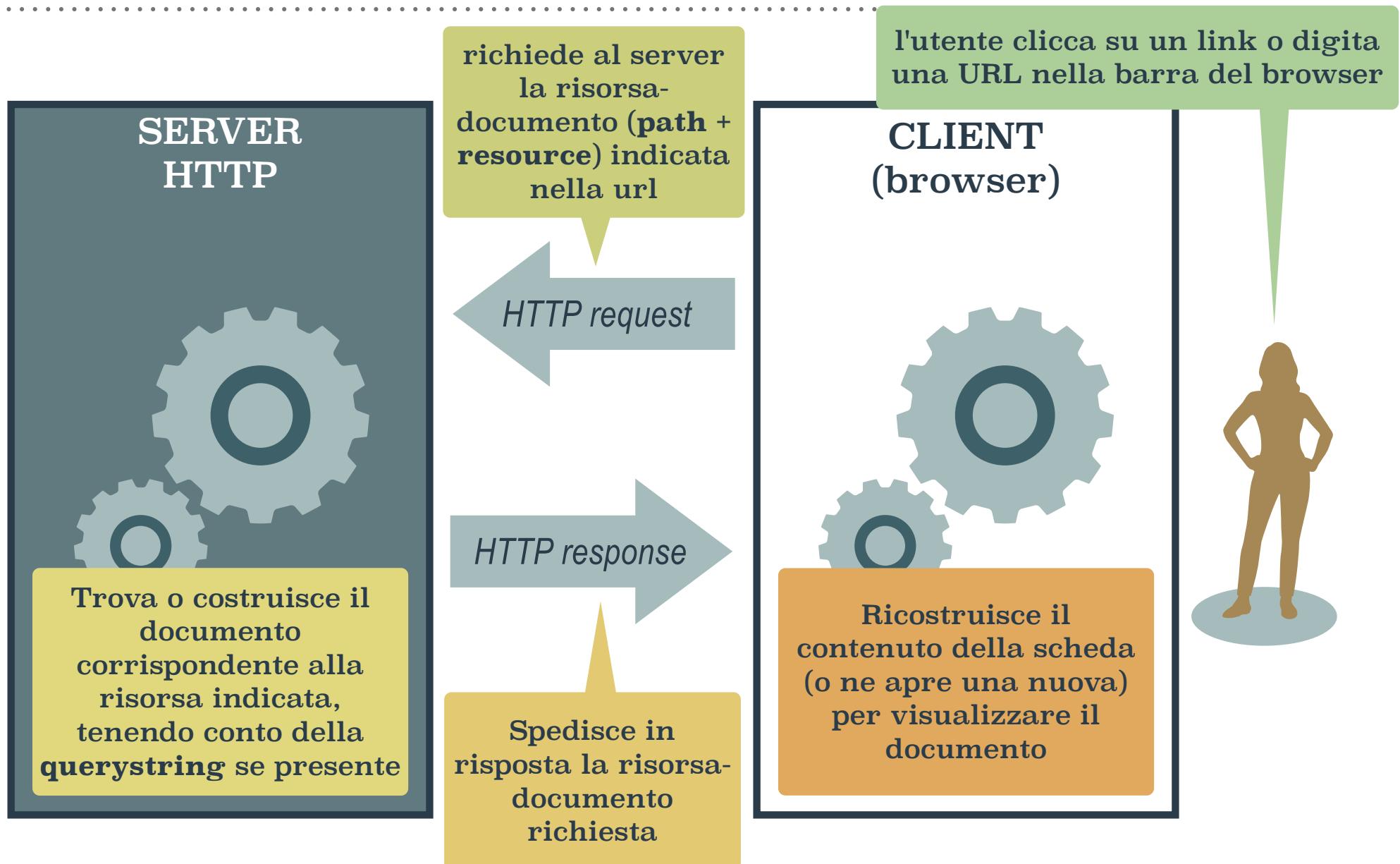
```
const http = require('http');

const hostname = "127.0.0.1";
const port = 8613;

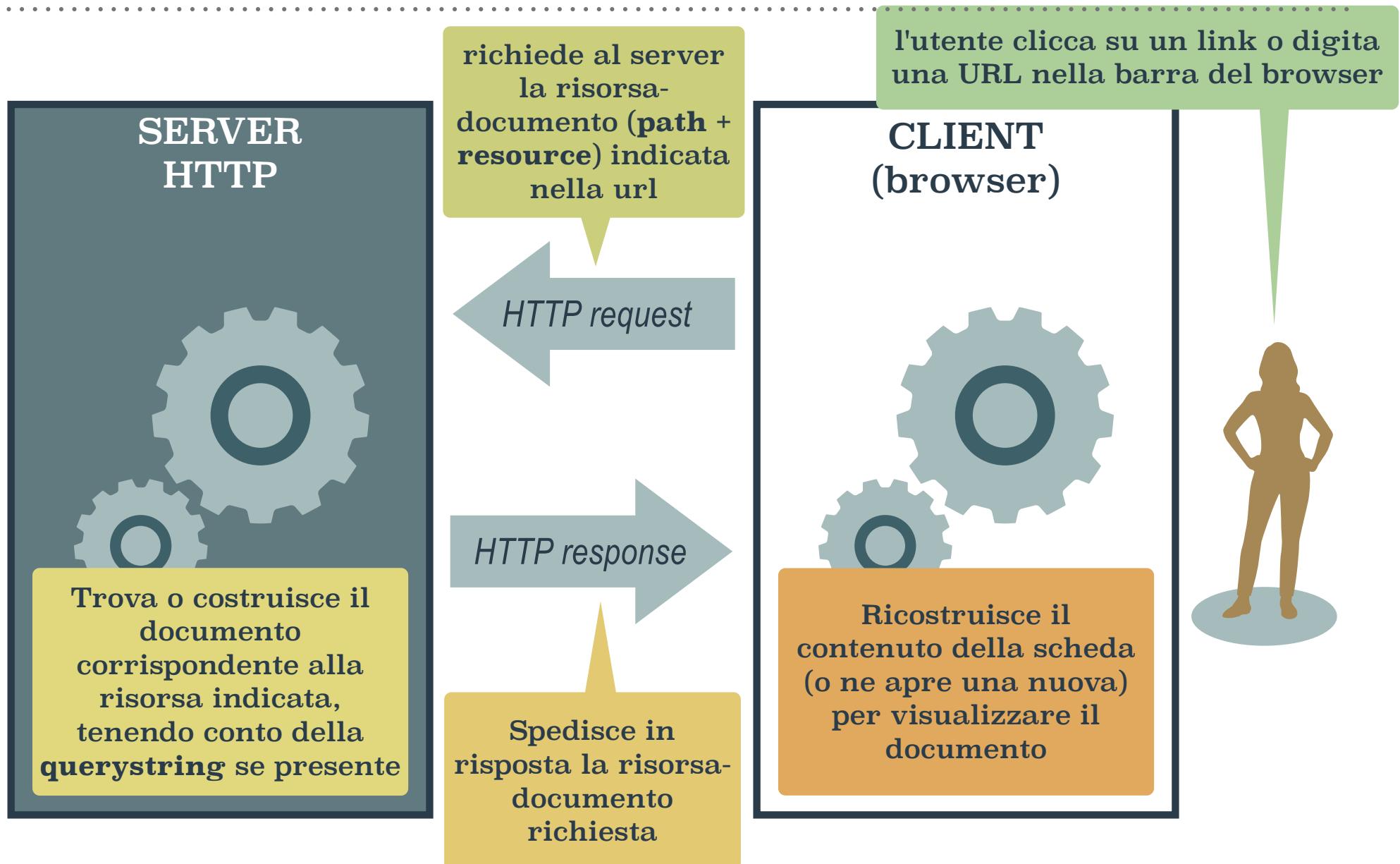
const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html');
    res.end(`<h1>Eccomi!</h1>
<p>Ho appena spedito un file html dal server al client!</p>`);
});

server.listen(port, hostname, () => {
    console.log("Server running at http://" + hostname + ":" + port);
});
```

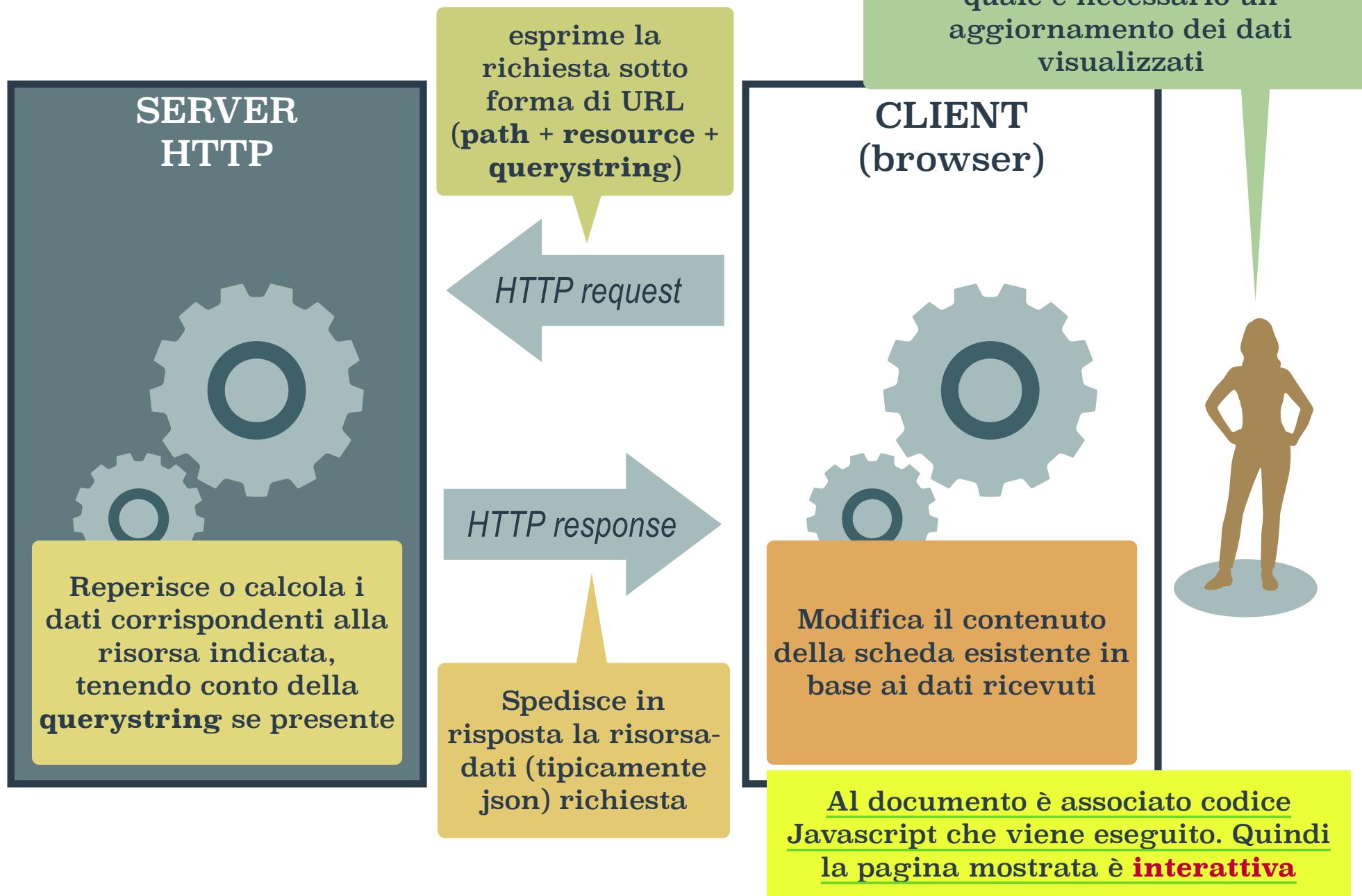
OTTENERE INFORMAZIONI (DATI, DOCUMENTI) DAL SERVER



OTTENERE INFORMAZIONI (DATI, DOCUMENTI) DAL SERVER



OTTENERE INFORMAZIONI (DATI, DOCUMENTI)



VARIANTE 1



esprime la richiesta sotto forma di URL
(**path + resource + querystring**)

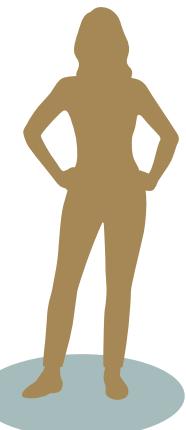
HTTP request

HTTP response

Spedisce in risposta **la porzione di documento elaborata**

l'utente effettua qualche interazione per soddisfare la quale è necessario un aggiornamento dei dati visualizzati

CLIENT
(browser)



Modifica il contenuto della scheda esistente **includendo la porzione di documento ricevuta**

Al documento è associato codice Javascript che viene eseguito. Quindi la pagina mostrata è **interattiva**

VARIANTE 2



esprime la richiesta sotto forma di URL (**path + resource + querystring**)

HTTP request

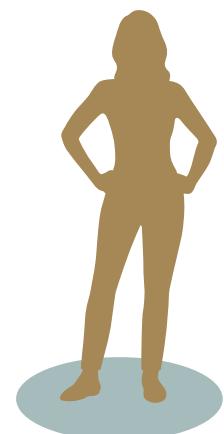
HTTP response

Spedisce in risposta il documento elaborato



Al documento è associato codice Javascript che viene eseguito. Quindi la pagina mostrata è **interattiva**

l'utente effettua qualche interazione per soddisfare la quale è necessario un aggiornamento dei dati visualizzati



VARIANTE 2



esprime la richiesta sotto forma di URL (**path + resource + querystring**)

HTTP request

HTTP response

Spedisce in risposta il documento elaborato



l'utente effettua qualche interazione per soddisfare la quale è necessario un aggiornamento dei dati visualizzati

Anche l'ambiente Javascript viene resettato, eventuali nuovi script vengono eseguiti.

HTTP GET

Il tipo di **HTTP request** più comune, necessario per ottenere risorse (documenti o dati), va sotto il nome di **GET**

Anatomia di base di una **HTTP request**:

- **method**: il tipo di richiesta, che fa capire al server qual è l'obiettivo
 - in questo caso il **method** è **GET** e fa capire al server che l'intenzione è **ottenere** delle risorse
- **url**: l'identificatore della risorsa che si vuole richiedere, secondo quanto già visto
 - **path**: la collocazione della risorsa (in un file system, in un db, in uno schema logico deciso dai progettisti del server... chi la richiede non deve sapere come verrà reperita fisicamente la risorsa, il path è un'astrazione, non ha necessariamente una controparte fisica)
 - **name**: il nome della risorsa; anche in questo caso può essere un'astrazione e non avere una corrispondenza reale; l'importante è che il server sappia come usarlo (in combinazione con l'eventuale **querystring**) per trovare/calcolare la risorsa desiderata
 - **querystring**: opzioni che permettono al server di individuare con precisione la risorsa richiesta, o di configurarne l'elaborazione.
- **headers**: intestazioni opzionali che permettono di fornire meta-information particolari sulla richiesta o sul client che la sta effettuando
 - per il momento tralasciamo questa parte

LA RISPOSTA DEL SERVER

Il server risponde compilando una **HTTP response**

Anatomia di base di una **HTTP response**:

- **status code**: un codice a tre cifre che identifica il tipo di risposta
 - la prima cifra può essere 1, 2, 3, 4 o 5
 - ai nostri fini è importante sapere che:
 - gli status code di tipo 2xx indicano che la richiesta ha avuto successo. Per le richieste **GET** il codice è tipicamente 200
 - gli status code di tipo 4xx indicano che la richiesta non è valida, perché è malformata, non corrisponde a una risorsa esistente, il client non ha i permessi per quella risorsa... Per le richieste **GET** i codici più frequenti sono 400 (bad request), 403 (forbidden), 404 (not found)
 - gli status code di tipo 5xx indicano che c'è stato un errore sul server che ha impedito di elaborare una risposta, a fronte di una richiesta almeno apparentemente valida. Per le richieste **GET** i codici più frequenti sono 500 (internal server error), 503 (service unavailable)
- **body**: il contenuto (documento, dati, file di vario tipo come immagini, video, pdf...)
- **headers**: intestazioni opzionali che permettono di fornire meta-information particolari sulla risposta o sul server che la sta fornendo
 - **Content-Type**, che comunica al client in cosa consiste il **body**
 - text/html, text/plain, application/json
 - image/jpeg, image/png, video/mp4, audio/mpeg, application/zip, application/pdf

SIMPLE SERVER

.....

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

const hostname = "127.0.0.1";
const port = 8613;

const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html');
    res.end(`<h1>Eccomi!</h1>
<p>Ho appena spedito un file html dal server al client!</p>`);
});

server.listen(port, hostname, () => {
    console.log("Server running at http://" + hostname + ":" + port);
});
```

MODULI IN STILE 'COMMONJS'

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

Node.js introduce un proprio stile per esportare/importare oggetti (inclusi funzioni e classi) fra diversi file .js (o da librerie, come in questo caso)

Per esportare un singolo oggetto da un proprio file **mycode.js**:

```
module.exports = oggettoDaEsportare;
```

Per usare l'oggetto in un altro file .js possiamo importarlo così:

```
const myStuff = require("./mycode");
```

Per esportare più oggetti da un proprio file **myothercode.js**:

```
exports.a = qualcheOggetto;
exports.b = qualcheAltroOggetto;
```

Per importare alcuni o tutti i mulipli oggetti esportati in un altro file .js:

```
const { a, b } = require("./myothercode");
```

CREARE UN SERVER IN NODE.JS

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

La libreria **http** crea un server come una **funzione di callback**.

La funzione prende come argomenti due oggetti che rappresentano la **Request** e la **Response**. Il primo oggetto può essere usato per ottenere informazioni sulla richiesta ricevuta. Il secondo viene usato per inviare al client una opportuna risposta.

```
server.listen(port, hostname, () => {  
  ...  
});
```

Attenzione: questa è un'altra callback che viene invocata quando il server si attiva correttamente

Il server viene attivato ponendolo in ascolto su un dato **indirizzo ip** e una **data porta**.

La **callback** precedentemente definita con **createServer** viene associata ad ogni arrivo di una **HttpRequest** all'**host** indicato sulla **porta** specificata

INVIO DI UNA RISPOSTA

```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html');
  res.end(`<h1>Eccomi!</h1>
<p>Ho appena spedito un file html dal server al client!</p>`);
});
```

impostiamo gli elementi di base di una **HttpResponse** e la inviamo.

statusCode comunica al client se la richiesta è andata a buon fine. Ci sono tantissimi statusCode, i più comuni sono 200 “OK” e 404 “Page Not Found”.

setHeader permette di impostare uno dei vari header delle **HttpResponse** che forniscono meta-informationi al client. L'header più comune è **Content-Type** che identifica il tipo di contenuto con un **MIME type**: in questo caso **text/html** (altri valori comuni: **text/plain**, **image/jpeg**, **image/png**, **application/json**, **video/mp4...**). Questo header viene usato dal client per sapere cosa fare con ciò che riceve.

end invia quanto specificato al client tramite **HttpResponse**, e chiude il flusso di output. Importante: **res** è un **write stream**: significa che, finché non lo chiudiamo, possiamo inviare dati su di esso. È però fondamentale chiuderlo, o il client resterà in attesa di informazioni. Con **end** facciamo entrambe le cose (invio e chiusura) in un colpo solo.

ESEMPIO - CONOSCERE LA URL DELLA RICHIESTA



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

const hostname = "127.0.0.1";
const port = 8613;

const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html');
    if (req.url === "/") {
        res.end(`<h1>Eccomi!</h1>
<p>Ho appena spedito un file html dal server al client!</p>`);
    } else if (req.url === "/pippo") {
        res.end(`<h1>Mi hai chiesto Pippo!</h1>`);
    } else {
        res.statusCode = 404;
        res.end("<h1>OPS!</h1>");
    }
});

server.listen(port, hostname, () => {
    console.log("Server running at http://" + hostname + ":" + port);
})
```

DIFFERENZIARE LA RISPOSTA SULLA BASE DELL'URL

```
if (req.url === "/") { ... }
```

req rappresenta la `HttpRequest` e **req.url** contiene la porzione della URL di richiesta **successiva** a protocollo e hostname.

ESEMPIO - SERVIRE UN DOCUMENTO HTML COSÌ COM'È

.....

```
const http = require('http');
|const fs = require('fs');| ←

const hostname = "127.0.0.1";
const port = 8613;

const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html');
|const read = fs.createReadStream("./doc.html");| ←
    read.pipe(res);
});

server.listen(port, hostname, () => {
    console.log("Server running at http://" + hostname + ":" + port);
})
```

ALCUNE NOTE SUL PACKAGE DI GESTIONE DEL FILE SYSTEM

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

fs è il package di Node.js che gestisce il *file system*.

Non abbiamo tempo per affrontare la gestione del file system in dettaglio, alcune informazioni di base per chi volesse approfondirlo:

- l'input e l'output su file sono gestiti da Node.js tramite il concetto di **stream** (input e output stream)
- le operazioni su file stream sono asincrone: i metodi che esso definisce (come vedremo in questo esempio) fanno partire l'operazione e ritornano immediatamente.
- se mi interessa sapere quando l'operazione è terminata e con quale risultato (ad esempio, i dati letti da un file) posso registrare una **callback** che verrà chiamata al momento opportuno.

INVIARE IL CONTENUTO DELL'HTML SULLA RESPONSE

```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html');
  const read = fs.createReadStream("./doc.html");
  read.pipe(res);
});
```

createReadStream crea un input (read) stream a partire da un file

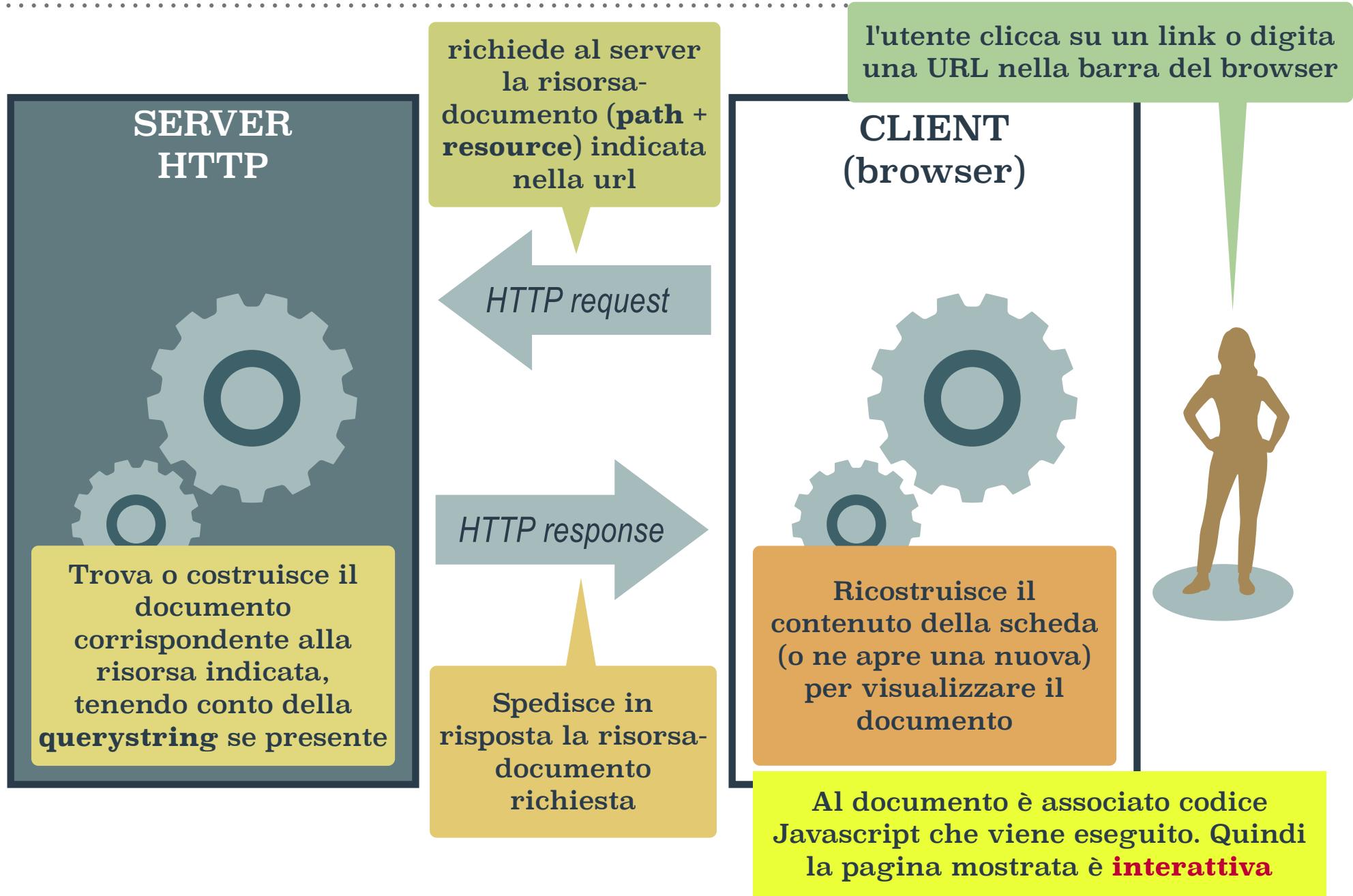
Quando un read stream viene letto solleva eventi in determinate circostanze: ad esempio quando si verifica un errore (evento **error**), quando termina la lettura (evento **end**), quando “consuma” un blocco di dati (evento **data**). E’ possibile registrare delle callback per questi eventi con il metodo **on**:

 `read.on("error", () => { /* do something with error */ });`

pipe inoltra il **read stream** su un **write stream** e poi lo chiude.

È completamente asincrona: ritorna immediatamente e, se non registriamo alcuna callback, il nostro server non deve più occuparsi di questo invio. Se anche fosse un file enorme il nostro server è pronto ad accettare nuove richieste.

INVIARE DATI AL SERVER



INVIARE DATI AL SERVER



la richiesta deve riportare sia la URL che i **dati** in questione

HTTP request

HTTP response

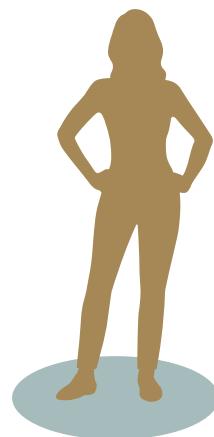
La URL viene utilizzata in questo caso per indicare la **destinazione dei dati e/o il tipo di elaborazione** da effettuare su di essi

Può inviare in risposta **dati, documenti** o altro, anche solo **se la richiesta è andata a buon fine**

CLIENT
(browser)

l'utente effettua qualche interazione per cui è necessario **inviare dati al server** perché li usi in qualche modo

Modifica il contenuto della scheda esistente in base all'**esito della richiesta** e alla risposta ricevuta



HTTP POST

Il tipo di HTTP **request** base per inviare dati al server è **POST**

Approfondiamo l'anatomia della **HTTP request** per le POST:

- **method**: il tipo di richiesta, che fa capire al server qual è l'obiettivo
 - in questo caso il **method** è **POST** e fa capire al server che **l'intenzione è inviare dati**
- **url**: l'identificatore della risorsa che si vuole richiedere, secondo quanto già visto
 - **path+name**: identificano la destinazione dei dati e/o il tipo di elaborazione che si vuole venga fatta su di essi; permette al server di capire quali operazioni deve compiere sui dati ricevuti
 - **querystring**: eventuali opzioni relative all'elaborazione/memorizzazione dei dati in questione.
http://mia.applicazione.org/utenti
http://mia.applicazione.org/utenti/numero-tel
http://mia.applicazione.org/documenti?uid=123
- **body**: il contenuto (dati) della richiesta
- **headers**: intestazioni opzionali che permettono di fornire meta-information particolari sulla richiesta o sul client che la sta effettuando
 - **Content-Type** diventa rilevante anche per la **request** perché permette di specificare il tipo di dati che si stanno inviando
 - **text/html**, **text/plain**, **application/json**, **application/x-www-form-urlencoded**
 - anche tipi di file se la richiesta consiste nell'upload di un file

ESEMPIO - STAMPARE IL CORPO DELLA RICHIESTA A CONSOLE

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

const hostname = "127.0.0.1";
const port = 8613;

const server = http.createServer((req, res) => {
  if (req.method === 'POST') {
    console.log(req.headers['content-type']);
    req.pipe(process.stdout);
    res.end();
  }
}) ;

server.listen(port, hostname, () => {
  console.log("Server running at http://" + hostname + ":" + port);
})
```

REQUEST COME INPUT STREAM

```
if (req.method === 'POST') {  
    console.log(req.headers['content-type']);  
    req.pipe(process.stdout);  
    res.end();  
}
```

req.method contiene il method della request

req.headers contiene tutti gli header come associazioni chiave/valore

quando la richiesta è di tipo **POST**, l'oggetto **req** è un **read (input) stream**

Può essere inviato con **pipe** su un **write (output) stream**, come abbiamo visto prima per inviare il file (input) sulla response (output).
process.stdout è il write stream di default per la console.

Anche qui è possibile catturare gli errori di lettura:

```
req.on("error", () => { /* do something with error */ });
```

ESEMPIO - SCRIVERE IL CORPO DELLA RICHIESTA IN UN FILE

.....

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

const hostname = "127.0.0.1";
const port = 8613;

const server = http.createServer((req, res) => {
  if (req.method === 'POST') {
    const myfile = fs.createWriteStream("request.txt");
    req.pipe(myfile);
    res.end();
  }
}) ;

server.listen(port, hostname, () => {
  console.log("Server running at http://" + hostname + ":" + port);
})
```



SCRIVERE SU FILE

```
const myfile = fs.createWriteStream("request.txt");
req.pipe(myfile);
```

createWriteStream crea un output (write) stream a partire da un file

Quando un write stream viene scritto solleva eventi in determinate circostanze: ad esempio quando si verifica un errore (evento **error**), quando lo scrivente segnala di aver terminato (evento **finish**). Anche in questo caso è possibile registrare delle callback con il metodo **on**:

```
myfile.on("finish", () => { /* do something when file has been written */ });
```

HTTP PUT, PATCH, DELETE

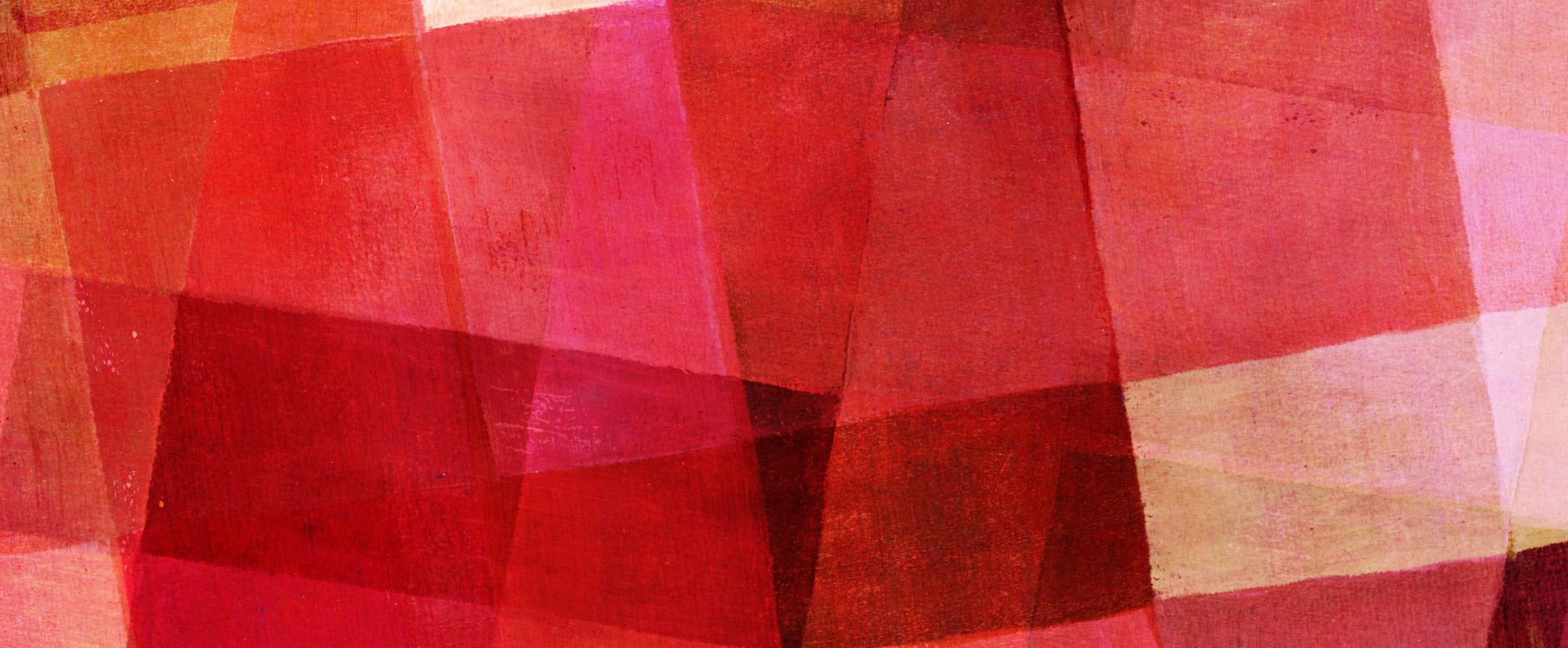
Non ci sono solo GET e POST!

Il **method** di una **request** distingue il tipo di elaborazione richiesto al server

- se usiamo solo GET e POST stiamo distinguendo la ricezione (GET) e l'invio (POST) di informazioni
- possiamo effettuare una distinzione più fine in base al **tipo di elaborazione** richiesto
 - **GET**: elaborazione volta a **ottenere** dei dati (es: l'elenco degli utenti)
 - **POST**: elaborazione volta a **inserire nuovi dati** (es: inserire un nuovo utente)
 - **PUT**: elaborazione volta a **reinserire un dato esistente** (es: sostituire completamente i dati di un utente presente nel sistema)
 - **PATCH**: elaborazione volta a **correggere un dato esistente** (es: modificare il numero di telefono di un utente)
 - **DELETE**: elaborazione volta a **eliminare un dato** (es: eliminare un utente)

`http://mia.applicazione.org/utenti?tel=011666555`

- sta al progettista del server stabilire quali **path+resource** (detti anche: **route**) sono accettati dal suo server e quali metodi sono attivi per ciascuno di essi
 - può scegliere di usare uno schema basato solo su GET/POST
 - oppure basato su tutti e 5 i method principali
- L'insieme delle **route** valide e dei **method** accettati per ciascuna di esse costituisce l'**API** del server



NODE.JS PER WEB APP

COS'È node ?

*È un ambiente di esecuzione per Javascript

- ◆ basato sul motore Javascript V8 sviluppato da Google per Chrome
- ◆ permette dunque di programmare in Javascript al di fuori di un browser

*Inoltre...

- ◆ Fornisce un insieme di moduli built-in (in parte OS-dependent) che facilitano alcuni compiti
 - in particolare è molto semplice *scrivere* un HTTP Server in Node.js
- ◆ Include **npm**
 - repo di moduli javascript open-source
 - tool di package management

QUANDO E PERCHÉ

Quando

Server-Side Web App Dev

Perché

Stesso linguaggio per client e server

Il codice sviluppato non deve essere distribuito e installato su un http server, ma diventa “parte” del server stesso.

Client-Side Web App Dev

Nella fase di sviluppo in locale, l’uso di Node.js semplifica debug, test e la gestione delle dipendenze

Cross-platform Desktop App Dev

Web Server in localhost con accesso al file system + Web Client = Desktop App

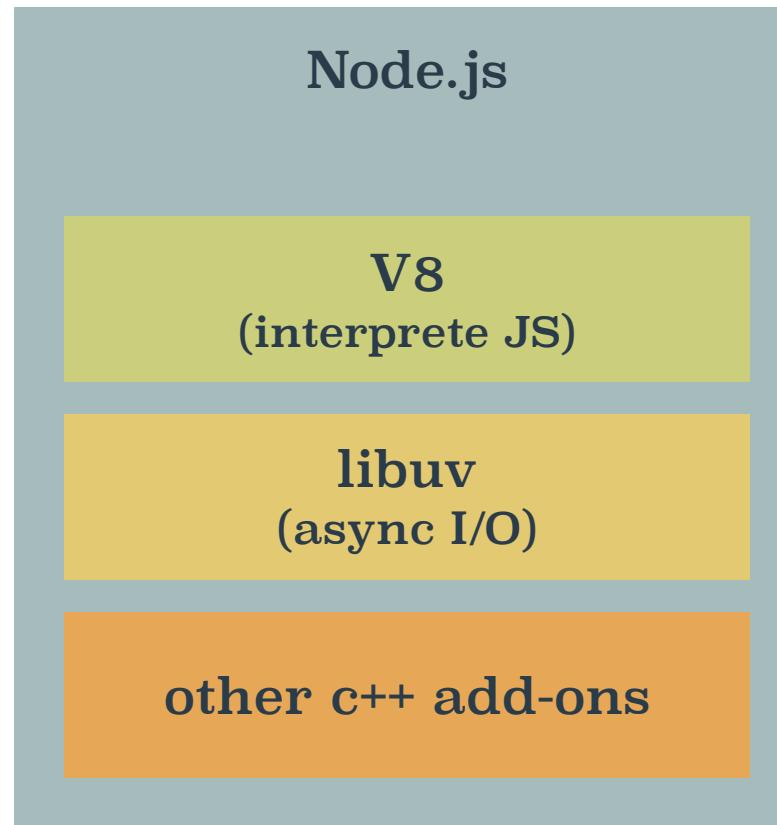
- ➡ Electron = Node.js + Chromium
- ➡ es.: VisualStudioCode, Atom, Skype

Cloud Dev

Diversi cloud provider supportano Node.js nei loro servizi FaaS e PaaS

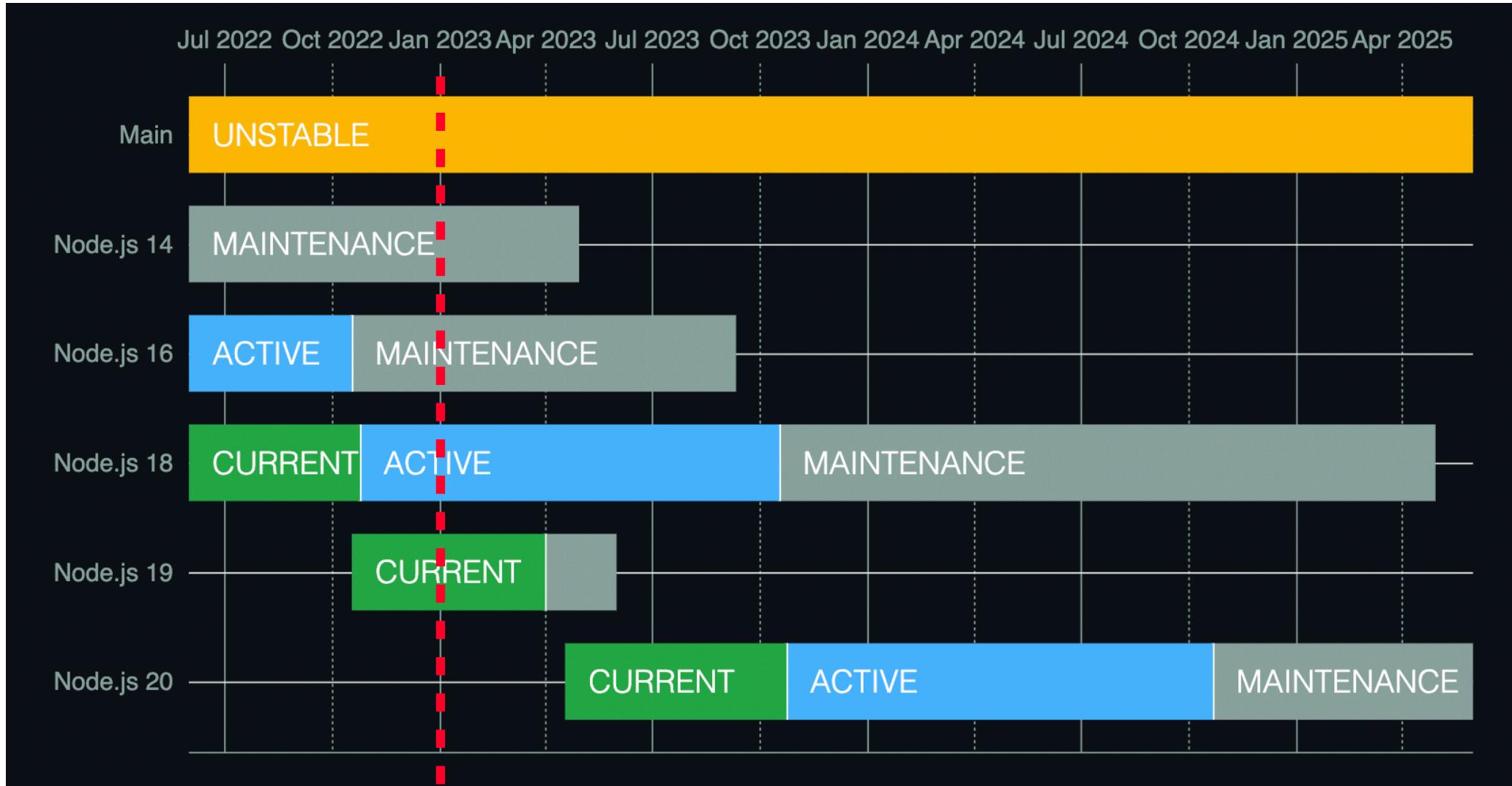
- ➡ es. AWS, Azure, Heroku

UN'OCCHIATA ALL'INTERNO



RELEASE DI NODE

.....



COME USARE NODE

REPL

> node

Read
Eval
Print
Loop

console per eseguire
interattivamente
istruzioni javascript

single-script

> node nomescript.js

esegue lo script usando il
motore di Node.js

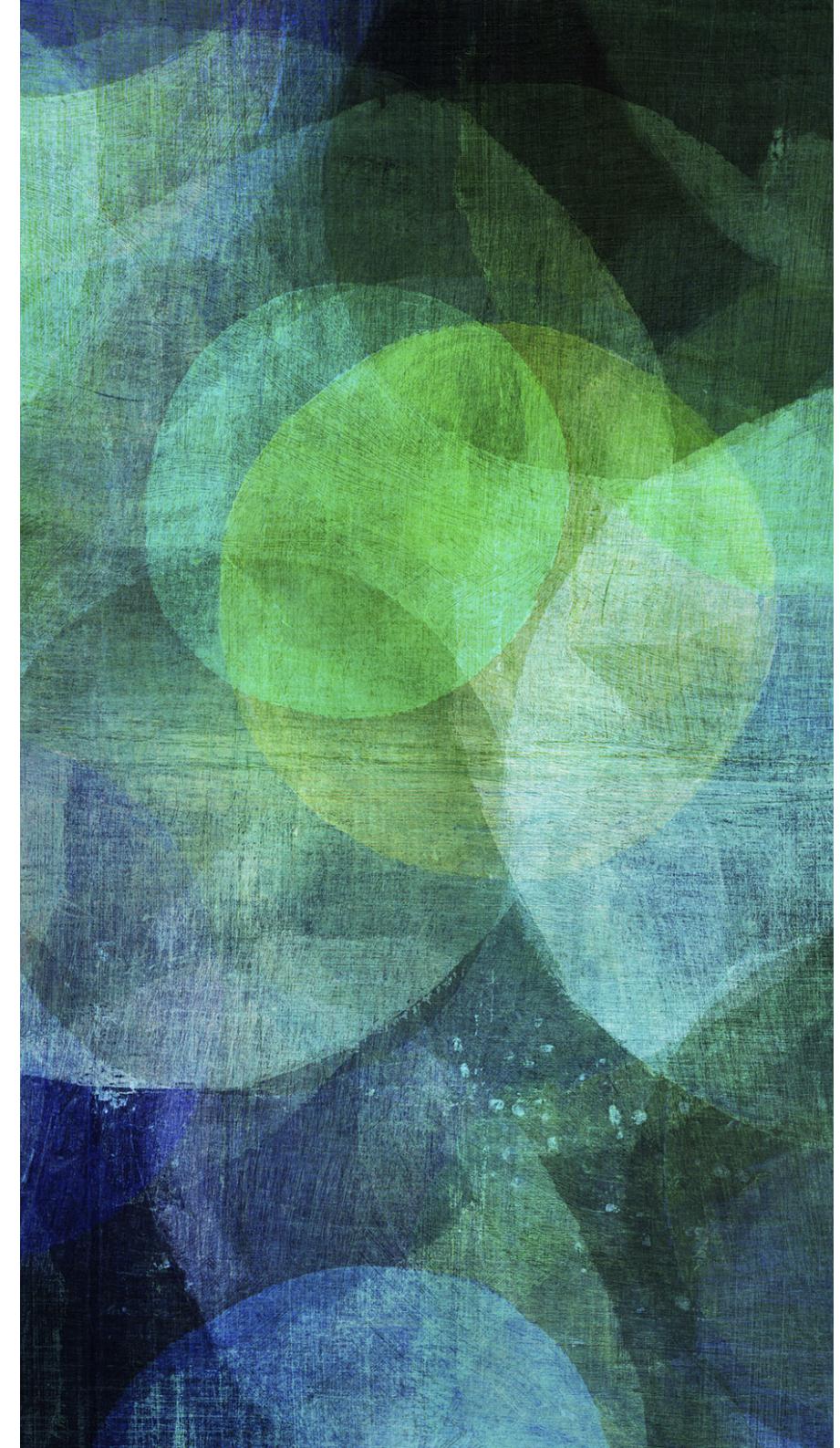
Project

> npm init

Un progetto è formato da
più file .js, dai moduli di
terze parti che essi usano,
e da un file 'package.json'
che specifica la
configurazione del
progetto. I progetti sono
gestiti tramite npm, 'node
package manager'

NPM NODE PACKAGE MANAGER

*Condivisione e riuso del codice
Gestione delle dipendenze*



COS'È E A COSA SERVE ?

*NPM = Node Package Manager

*Obiettivo originale

- ◆ condivisione del codice javascript fra sviluppatori
- ◆ riuso del codice javascript attraverso diversi progetti
- ◆ il concetto base è quello di **package**
 - anche detto **modulo**
 - un modulo o package è una **cartella** contenente uno o più file **javascript** che realizzano un ben preciso e coeso insieme di funzionalità
 - può essere una **singola funzionalità** molto piccola, o un'intera **libreria**, o addirittura un **framework**
- ◆ i package sono distribuiti tramite il sito npmjs.com
- ◆ upload e download avvengono principalmente tramite il command-line tool **npm**

*Non solo in Node.js

- ◆ Sebbene **npm** sia nato come parte di Node, viene usato per condividere e scaricare package per qualsiasi tipo di progetto Javascript



npm E LA GESTIONE DELLE DIPENDENZE

* **npm** ha cambiato (in meglio!) la vita ai programmatori javascript

- ◆ la modularizzazione è ormai estremamente elevata
- ◆ ogni modulo dipende da altri moduli che dipendono da altri moduli...
 - la potenza di questo meccanismo è solo pari alla difficoltà di gestione
 - cosa succede se un package molto utilizzato viene ritirato dai suoi sviluppatori?
 - <https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm.html>

* **npm** viene usato anche per gestire le **dipendenze** di un progetto

- ◆ mantiene un file **package.json** che registra informazioni sul progetto
 - fra cui le dipendenze di primo livello
- ◆ mantiene l'**albero** delle sottodipendenze
- ◆ con un solo comando permette di scaricare e installare tutti i moduli richiesti da un progetto
 - in questo modo nel distribuire il progetto si può evitare di includere anche i moduli da esso utilizzati

* lo strumento principale è la CLI (command line interface) di **npm**

- ◆ npm [comando] [opzioni] [parametri]

NPM: INSTALLARE UN PACKAGE NEL MIO PROGETTO

***npm install [package]** installa **localmente** un package per il progetto corrente

***npm install -g [package]** installa un package **globalmente** ossia a livello di sistema; utile per strumenti da linea di comando che si usano durante lo sviluppo

***npm install -D [package]** installa **localmente** un package per il progetto corrente con opzione “development” ossia da usare solo in fase di sviluppo e non per l’applicazione finale.

- ◆ si possono installare strumenti da linea di comando anche localmente, dovranno però essere eseguiti non “direttamente” bensì tramite il comando **npx** (**npm execute**)

***npm install** (re)installa tutte le dipendenze **locali** dichiarate in package.json

SEMANTIC VERSIONING (SEMVER)

.....

4 . 17 . 1

major
breaking changes

minor
backward compatible

patch
bug fixes

Ok la 1.2.3 o qualsiasi **patch** posteriore
Ossia: 1.2.x con x maggiore o uguale a 3

~1.2.3

Ok la 1.2.3 o qualsiasi **minor** posteriore
Ossia: 1.y.x con y maggiore o uguale a 2 e x qualunque

^1.2.3

Ok solo la versione 1.2.3

=1.2.3

***npm update** aggiorna tutte le dipendenze dichiarate in
package.json nel rispetto delle dichiarazioni **semver**

NPM: RECAP COMANDI UTILI

*inizializzazione:

- **npm init** inizializza un nuovo progetto

*installazione:

- **npm install [package]** installa un package per il progetto corrente
- **npm install -g [package]** installa un package a livello di sistema
- **npm install -D [package]** installa un package per il progetto corrente con opzione “development” ossia da usare solo in fase di sviluppo e non per l’applicazione finale
- **npm install** installa tutte le dipendenze dichiarate in package.json

*aggiornamento:

- **npm update** aggiorna tutti i package alla versione più recente compatibile con quanto dichiarato in package.json

*esecuzione:

- **npm start** esegue (se presente) lo script “start” dichiarato in package.json, voce *scripts*
- **npm run xxx** esegue (se presente) un qualunque script chiamato “xxx” dichiarato in package.json, voce *scripts*

*varie ed eventuali:

- **npm list** elenca i moduli installati con le rispettive versioni