Vitaletti

Docker

Docker è una tecnologia che automatizza istallazione e configurazione, la maggior parte delle app sono Cloud,cioè mi garantisce elasticità di servizio, portabilità tra ambienti di sviluppo (per garantire portabilità devi avere omologazione certi standard – non mi porto dietro il SO), lock-in

Divido l'applicazione in microservizi, in modo tale da rendere il tutto scalabile rispetto ad applicazione monolitica. Vantaggio sta nel non dover riscrivere tutto un codice (o anche ricompilarlo) ogni volta ma modificare unicamente il microservizio specifico, inoltre in questo modo non ho un unico punto di rottura.

Nel mondo moderno ci sono quindi 3 approcci indipendenti delle infrastrutture:

1) Tradizionale

- *Costi molto alti*: perché ogni volta devo comprare un nuovo oggetto da far girare sulla macchina

- *Lento lo sviluppo:* perché dopo aver finito l'istallazione e dopo aver sviluppato tutto,se voglio installarlo su un altra macchina devo vedere se risulta essere compatibile. Se cosi non fosse devo modificarlo e riprovare finché non risulti essere compatibile.

2) Tradizionale Moderno – Virtualizzazione

- Benefici
  - Server fisico può contenere applicazioni multiple
  - Ogni app gira su virtual machine
  - Una fisica VM viene divisa in VM
  - Maggiore scalabilità rispetto all'approccio tradizionale

- Svantaggi
  - Un intero Guest OS che genera spreco di risorse
  - Più macchine VM girano, più risorse ne hanno bisogno
  - Ogni VM richiede allocazione CPU

    - Portabilità delle applicazioni non è garantita

3) Container

- Virtualizzando solamente il sistema operativo e le componenti necessarie all'esecuzione delle applicazioni (librerie e file binari), invece che l'intera macchina, i tempi di avvio si riducono notevolmente rispetto quelli di una VM

- Container consentono agli sviluppatori di suddividere ulteriormente le risorse in microservizi, garantendo maggior controllo sull'eseguibilità delle applicazioni e miglioramento delle performance generali.

- Container non devono attivare SO

- Vantaggi
    - Portabilità
    - Efficienza
    - Velocità

Se nella VM ho un'infrastruttura fisica, un Hypervisor e vari guest; in Docker utilizzo "container" che hanno solo le librerie per far girare app.

DOCKER BASIC

Immagine: Classe
Container: Istanza di una classe
Volume: Astrazione dello storage
Docker Compose: descrive più istanze di un container
DockerFile: descrive sola istanza del container

```
docker build -t friendlyhello .  # Create image using this directory's Dockerfile
docker run -p 4000:80 friendlyhello  # Run "friendlyname" mapping port 4000 to 80
docker run -d -p 4000:80 friendlyhello         # Same thing, but in detached mode
docker container ls                                # List all running containers
docker container ls -a             # List all containers, even those not running
docker container stop <hash>           # Gracefully stop the specified container
docker container kill <hash>         # Force shutdown of the specified container
docker container rm <hash>        # Remove specified container from this machine
docker container rm $(docker container ls -a -q)         # Remove all containers
docker image ls -a                             # List all images on this machine
docker image rm <image id>            # Remove specified image from this machine
docker image rm $(docker image ls -a -q)   # Remove all images from this machine
docker login             # Log in this CLI session using your Docker credentials
docker tag <image> username/repository:tag  # Tag <image> for upload to registry
docker push username/repository:tag            # Upload tagged image to registry
docker run username/repository:tag                   # Run image from a registry
```

DOCKER NETWORK

La rete di Docker è costituita da Drivers:

- *Bridge* è la rete di default che viene creata appena avvii Docker e i container avviati si ollegano ad essa se non diversamente specificato. Bridge consente ai container, connessi ad essa, di comunicare tra di loro (per esempio NodeJS, CouchDB e Nginx che si trovano in container isolati).

- *Overlay* è equivalente al bridge ma i container non sono posti su un unica macchina ma sparsi nel mondo di internet

- *Host*

- *Macvlan* consentono di assegnare un indirizzo MAC a un contenitore, facendolo apparire come un dispositivo fisico sulla rete. Questo implica che vi è una virtualizzazione anche dell'indirizzo MAC

Riepilogo

- **User-define bridge networks** sono le migliori quando sono necessari più container per comunicare sullo stesso host Docker.

  •**Host networks** sono le migliori quando lo stack di rete non deve essere isolato dall'host Docker, ma si desidera isolare altri aspetti del contenitore.

  •**Overlay networks** sono le migliori quando hai bisogno di container in esecuzione su diversi host Docker per comunicare o quando più applicazioni lavorano insieme utilizzando i servizi di swarm.

  •**Macvlan newtworks** sono le migliori quando si **esegue la** migrazione da un'installazione di VM o si richiede che i contenitori **sembrino** host fisici sulla rete, ognuno con un indirizzo MAC univoco.

DOCKER SWARM

Uno swarm è costuito da più Docker host che lavorano in modalità swarm e fungono sia da manger che da worker. In parole povere Docker swarm è un cluster di macchine che runnano docker; il senso è che vai a far eseguire le tua applicazione a diverse macchine e non più solo in locale da te stesso.

Quando si va a creare un servizio si va a definire lo stato ottimale (numero di repliche, risorse di rete e di archiviazione disponibili, porte che il servizio espone al mondo esterno e altro).

Docker funziona per mantenere lo stato desiderato. Ad esempio, se un nodo di lavoro diventa non disponibile, Docker pianifica le attività del nodo su altri nodi.

Nodi

Un **nodo** è un'istanza del Docker engine che partecipa allo swarm. È possibile eseguire uno o più nodi su un singolo computer fisico o su un server cloud, ma in genere le distribuzioni di swarm di produzione includono nodi Docker distribuiti su più macchine fisiche e cloud.

Per distribuire l'applicazione su uno swarm, si invia una specifica di servizio a un **nodo magager.** Il nodo manager invia unità di lavoro, chiamate task, ai nodi worker.

I nodi di lavoro ricevono ed eseguono attività inviate dai nodi manager. Il nodo worker notifica al nodo manager lo stato corrente delle attività assegnate in modo che il manager possa mantenere lo stato desiderato di ciascun lavoratore.

Servizi

Il servizio è l’attivita da eseguire sul manager o sui nodi worker. Quando si crea un servizio, si specifica l'immagine del container da utilizzare e quali comandi eseguire all'interno dei container in esecuzione.

Nei **servizi replicati**, il swarm manager distribuisce un numero specifico di attività di replica tra i nodi in base alla scala impostata nello stato desiderato.

Per **servizi globali**, lo swarm esegue un'attività per il servizio su ogni nodo disponibile nel cluster.

```
docker swarm init
```

→ Sul terminale per abilitare la modalità swarm e trasforma la tua macchina corrente in un swarm manager

```
docker swarm join
```

→ Esegui su altre macchine per farli entrare nello swarm come workers

```
docker-machine create --driver virtualbox myvm1
docker-machine create --driver virtualbox myvm2
```

→ Crea un paio di VM usando docker-machine, usando il driver VirtualBox

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ERRORS
myvm1   -        virtualbox   Running   tcp://192.168.99.100:2376           v17.06.2-ce
myvm2   -        virtualbox   Running   tcp://192.168.99.101:2376           v17.06.2-ce
```

→ Visualizzo le macchine

```
$ docker-machine ssh myvm1 "docker swarm init --advertise-addr <myvm1 ip>"
Swarm initialized: current node <node ID> is now a manager.

To add a worker to this swarm, run the following command:

  docker swarm join \
  --token <token> \
  <myvm ip>:<port>

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

→ Setto la prima macchina come swarm manager e la seconda come worker sulla porta 2377 che è la porta di default per docker swarm

```
docker stack deploy -c docker-compose.yml getstartedlab
```

→ Comando esegui per distribuire l’app, un singolo stack di servizi in esecuzione su un singolo host

N.B

Per aggiungere più servizi allo stack, li inserisci nel tuo file di Compose

Docker Stack: Uno stack è un gruppo di servizi correlati che condividono le dipendenze e possono essere orchestrati e ridimensionati. Un singolo stack è in grado di definire e coordinare le funzionalità di un'intera applicazione.

Socket e HTTP

Un socket è un endpoint di un collegamento di comunicazione bidirezionale tra due programmi in esecuzione sulla rete.

Un socket è associato a un numero di porta in modo che il livello TCP possa identificare l'applicazione a cui i dati sono destinati a essere inviati. Un endpoint è una combinazione di un indirizzo IP e un numero di porta.

Ogni connessione TCP può essere identificata in modo univoco dai suoi due endpoint. In questo modo puoi avere più connessioni tra l'host e il server.

Nginx – Web Server (F)

Un web server genera pagine statiche (file che vengono prelevati dal server e visualizzati sul browser), mentre un Application Server gestisce pagine dinamiche, cioè le pagine vengono “costruite” dinamicamente al tempo della chiamata (anche se può gestire anche quelle statiche).

La comunicazione tra web server e client avviene tramite il protocollo HTTP, che utilizza TCP sulla porta 80 per trasferire le risorse richieste. Il problema di un web server è che se dovesse ricevere una richiesta di tipo POST non può eseguirla, ma bisogna utilizzare un application server.

Un esempio di web server è NGINX. Quest’ultimo anziché creare nuovi processi per ogni richiesta web, utilizza un approccio asincrono basato su eventi nella gestione delle risorse in modo da ottenere prestazioni più prevedibili sotto stress, a differenza di Apache che invece utilizza un approccio orientato ai thread o processi nella gestione delle richieste.

Con NGINX, un processo “master” può controllare più processi “worker” i quali eseguono l’elaborazione effettiva in maniera indipendente l’uno con l’altro grazie all’asincronismo di NGINX che lo rende leggero e ad alte prestazioni.

Nodejs – Application Server

Node.js è un Application Server concepito per implementare la logica ma anche offrire servizi per l’interfaccia, in particolare utilizza framework di riferimento REST, inoltre consente quindi di scrivere applicazioni lato server utilizzando Javascript con un **modello asincrono basato sugli eventi non bloccanti,** andando cosi

a non sfruttare il classico modello basato su processi o thread concorrenti, utilizzato dai classici server.

Il modello event-driven, o “programmazione ad eventi”, si basa su un concetto piuttosto semplice: si lancia una azione quando accade qualcosa. Ogni azione quindi risulta asincrona a differenza dei pattern di programmazione più comune in cui una azione succede ad un’altra solo dopo che essa è stata completata.

L’efficienza dipenderebbe dal considerare che le azioni tipicamente effettuate riguardano il **networking**, ambito nel quale capita spesso di lanciare richieste e di rimanere in attesa di risposte che arrivano dopo un tempo relativamente lungo.

Grazie al **comportamento asincrono**, durante le attese di una certa azione si può gestire qualcos’altro. Per avere la sincronia sono necessarie CALLBACK annidate una dentro l’altra, in modo che per esempio, la funzione più interna venga generata solo dopo quella prima di lei.

Promises

Le Promises sono istanze della classe Promise e rappresentano operazioni che non sono state ancora completate, ma lo saranno in futuro.

Per creare una Promise abbiamo una funzione di callback con 2 argomenti fondamentali che ritorna l’esito della promessa (mantenuta oppure no) (resolve o reject)

Esempio creazione di una Promise:

```
let promiseToCleanTheRoom= new
Promise(function(resolve,reject){
	let isClean=false;
	if(isClean)
		resolve(“Clean”);
	else
		reject(“not Clean”);
});
```

Da notare che resolve e reject sono due funzioni implementate nelle librerie Javascript

```
//stampa il contenuto della Promise
promiseToCleanTheRoom.then(function(fromResolve){
        console.log("The Room is"+fromResolve);
})catch(function(fromReject){
        console.log("The Room is"+fromReject);
});
```

Rest

Rest è uno stile architetturale e non uno standard. RESTful viene in genere utilizzato per fare riferimento a servizi Web che implementano l'architettura REST.

Rest riguarda come rappresentare le risorse (identificate attraverso URI – come accedo, dove,con cosa,e a cosa sono interessato ad accedere), serve per manipolare le risorse, quindi un modo facile e flessibile di scrivere un web service.

Principi Implementazione REST

1) *Architettura Client/Server*: che rende i vari servizi implementati indipendenti l'uno dall'altro.

2) *Stateless:* vuol dire che non conserva lo stato/memoria del processo. Utilizziamo quindi, come in HTTP usiamo i Cookie (info che tracciano il comportamento dell'utente sul web)

3) *Cache*: utilizzo del caching, cioè se il contenuto che ho,è ancora valido, è inutile che vado alla sorgente

4) *Layered* System: trasparenza nell'interagire con server che a sua volta ha molti layer

5) *Code on demand:* una risorsa può essere codice

Rest è stateless, non conserva lo stato del processo,ma io voglio accedere allo stato (prendendolo o cambiandolo)

- Rappresentazione della risorsa con JSON
- Gestire stato risorsa (per esempio con HTTP)

Tramite i metodi REST: POST,GET,PUT,DELETE
posso andare a prendere le risorse "in giro" per il Web (Google,Facebook..) e li conpongo tra loro nell'Application Server.

In conclusione, REST ci consente,non solo di prendere/ottenere risorse, ma anche di manipolarle, quindi posso cambiarne lo stato.

Rest vs Crud

CRUD è l'acronimo di CREATE, READ, UPDATE, DELETE. Questi formano i comandi standard del database che sono alla base di CRUD.

Ad esempio, un acquirente su un sito eCommerce può CREARE un account, aggiornare le informazioni sull'account e CANCELLARE le cose da un carrello.

Questo vuol dire che REST è incentrato sulle risorse, CRUD è un ciclo pensato per mantenere record in un’impostazione di database. REST stabilisce una mappatura uno a uno tra le tipiche CRUD e i metodi HTTP.

Rest vs Soap

Soap utilizzato per lo scambio di messaggi tra componenti software secondo le regole di sintassi di XML. Soap è un protocollo, Rest è un framework.

La differenza principale tra SOAP e REST è il grado di accoppiamento tra implementazioni client e server. Un client SOAP funziona come un'applicazione desktop personalizzata, strettamente associata al server. C'è un contratto rigido tra il client e il server e non funziona più nulla se uno di questi cambia.

Un client REST è più simile a un browser. È un client generico che utilizza una interfaccia con metodi standardizzati che restituisce dati in formati più o meno standard.

REST è indipendente dal protocollo. Non è accoppiato necessariamente con HTTP. E' possibile seguire, ad esempio un link ftp da un sito web. Un'applicazione REST può utilizzare qualsiasi protocollo per il quale esiste uno schema standardizzato di richiesta URI.

Soap mette in risalto il concetto di servizio, invece il REST mette in risalto il concetto di risorsa. Un web service basato sul soap

espone un insieme di metodi richiamabili da remoto da parte di un client, mentre il REST è custode di un insieme di risorse sulle quali in client può chiedere le operazioni canoniche del protocollo HTTP.

Esempio Soap è più educato di REST.

REST : GET Coffee → diretto ma potrebbe non arrivarti il caffè che ti aspetti

SOAP: quello che voglio chiederti appartiene al Web Server "Coffee" (specfiche coffee)

XML vs JSON

XML e JSON sono i due formati più comuni per lo scambio di dati nel Web oggi.
XML è un linguaggio di markup è composto di istruzioni, definite tag o marcatori, che descrivono la struttura e la forma di un documento ed è possibile inserire i metadati nei tag sotto forma di attributi.
Uno dei vantaggi più significativi che XML ha su JSON è la sua capacità di comunicare contenuti misti, cioè stringhe che contengono markup strutturati.

JSON è uno strumento di serializzazione dati, utilizza coppie nome / valore, delineate in modo conciso da "{ "E"} "per gli oggetti," ["e"] "per gli array,", "per separare le coppie e": "per separare il nome dal valore.
JSON risulta essere meno prolisso di XML, perché XML richiede tag di apertura e chiusura e inoltre,viene tilizzato dalla maggior parte dei Service Provider.

CouchDB – Database - No SQL vs SQL

CouchDB è un database non relazionale (NOSQL). A differenza dei database relazionali **CouchDB non possiede il concetto di schema, ma di collezioni di documenti.**

Ogni documento in ogni database può essere strutturato in modo diverso dagli altri, cosa che invece non è possibile nei database relazionali (SQL) dove tutto ruota intorno al concetto di tabella

(formata da righe e colonne) e che tra le varie tabelle di un database possono esistere alcune relazioni.

Esempio di relazione : Una riga di una tabella A può fare riferimento ad un'altra riga di un'altra tabella B, e ciò può essere espresso inserendo la chiave primaria (indice che permette di riconoscere univocamente quella riga rispetto a tutte le altre) della riga di B tra i dati di quella di B.

CouchDB utilizza il protocollo HTTP e il formato dati JSON.
Le uniche chiavi obbligatorie sono *_id* (serve per identificare univocamente il documento) e *_rev* (utilizzato per la gestione di revisioni/modifiche), _rev viene aggiornato dopo ogni modifica.

Vantaggi database NOSQL

- *Elevata velocità computazionale,* anche al crescere del volume dei dati, legata alla mancanza di operazioni di aggregazione dei dati (no join)
- *Riduzione significativa dei tempi di sviluppo,* grazie alla definizione di logiche di lettura dati molto più semplici rispetto a quelle da scrivere con database relazionali

- *Supporto per la scalabilità orizzontale,* garantire tempi di risposta al crescere del carico con l'aggiunta di nuovi server
- *Elevato livello di disponibilità del servizio*

- *Schemaless,* non vi è necessità di una definizione formale e rigida del suo contenuto.

OAuth

OAuth 2 è un framework di autorizzazione (!= autenticazione, individuare un utente) che permette ad applicazioni di chiamare in modo sicuro ed autorizzato API messe a disposizione da un servizio Web. Permette di accedere alle risorse protette di un utente senza che esso debba condividere le sue credenziali (Una volta autenticato l'utente deve decidere i servizi ai quali è autorizzato ad accedere).

Funziona delegando l'autenticazione dell'utente al servizio che ospita l'account utente e autorizzando le applicazioni di terzi ad

accedere all'account utente proteggendo contemporaneamente le sue credenziali.

Per esempio permette all'utente di dare ad un sito, chiamato consumer, l'accesso alle sue informazioni presenti su un altro sito, detto service provider, senza condividere la sua identità.

L'idea di base è quella di autorizzare terze parti a gestire documenti privati senza condividere la password. La condivisione della password infatti presenta molti limiti a livello di sicurezza. OAuth è nato quindi con il presupposto di garantire l'accesso delegato ad un client specifico per determinate risorse sul server per un tempo limitato, con possibilità di revoca.

ESEMPIO. Entro dentro printfast.com, che detiene il servizio, voglio utilizzare l'App Picasa per trasformare le foto che ho su printfast.com in bianco e nero. Se non utilizzo OAuth e fornendo le mie credenziali per accedere ai servizi di Picasa in automatico le fornirei anche a printast.com -> PROBLEMA DI SICUREZZA

Con OAuth il client (printfast.com) chiede l'accesso alle risorse in nome dell'utente (non mi trovo più in printfast.com ma direttamente su picasa.com) che lo autorizza loggandosi a Picasa.com, il quale restituisce al client un token per accedere ai dati/servizi richiesti.

(Client: Printfast, Resource Owner: utente, Resource Server: Picasa, Authorization server: Google)

Ruoli

Proprietario Risorse: utente, chi autorizza un'App ad accedere al proprio account. L'accesso dell'applicazione all'account dell'utente è limitato allo "scopo" dell'autorizzazione concessa (ad esempio, accesso in lettura o scrittura).
Resource/Authorization Server (API): Il Resource Server ospita gli account utente protetti e l'Authorization server verifica l'identità dell'utente e poi invia i token di accesso all'applicazione. (Dal punto di vista dello sviluppatore dell'applicazione, l'API di un servizio soddisfa entrambi i ruoli dell'Authorization e del Resource server. Faremo riferimento a entrambi questi ruoli combinati, come il ruolo Servizio o API).

Client Application: Il Client è l’App che vuole accedere all’account dell’utente. Prima che possa farlo, deve essere autorizzato dall'utente e l'autorizzazione deve essere convalidata dall'API (Resource/Authorization server).

Flow

1. L’applicazione richiede l’autorizzazione all’utente per accedere alle risorse del servizio,

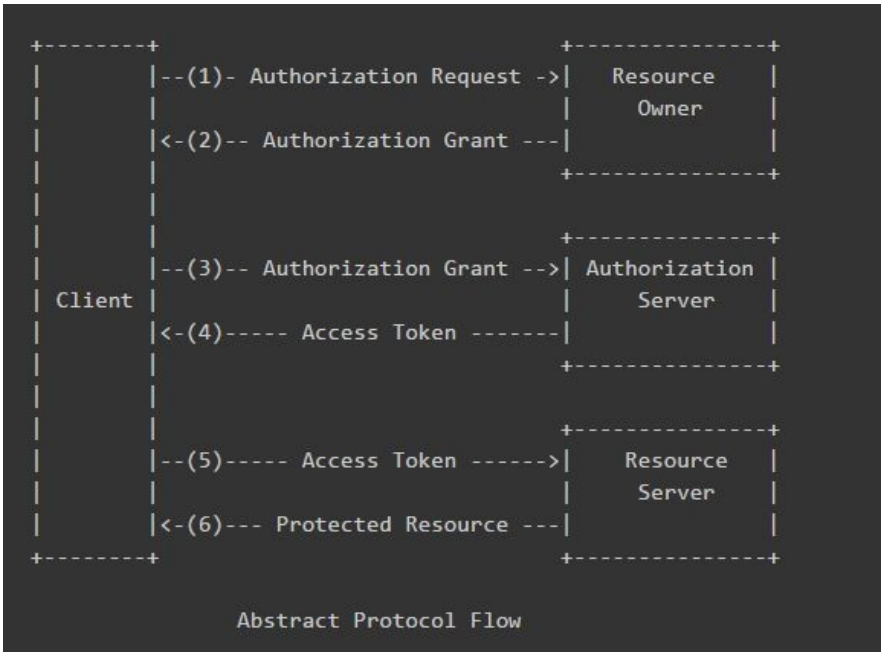
2. Se l’utente ha autorizzato la richiesta, l’App riceve un’Authorization Grant (concessione di autorizzazione)

3. L’App richiede un token di accesso all’authorization server (API) presentando l’autenticazione della propria identità e l’authorization Grant.

4. Se l’identità dell’applicazione è autenticata e l’authorization Grant è valido, l’authorization server (API) emette un token di accesso all’App. L’autorizzazione è completata.

5. L'applicazione richiede la risorsa dal Resource Server (API) e presenta il token di accesso per l'autenticazione

6. Se il token di accesso è valido, il Resource Server (API) serve la risorsa all'applicazione



Le prime quattro fasi riguardano l'ottenimento di un Authorization Grant e un token di accesso. Il tipo di Authorization Grant dipende dal metodo utilizzato dall'applicazione per richiedere l'autorizzazione e dai Grant types supportati dall'API.

OAuth 2 definisce quattro grant types, ciascuno dei quali è utile in diversi casi:
1. *Codice di autorizzazione*: utilizzato con applicazioni lato server
2. *Implicito*: utilizzato con le app mobili o le applicazioni Web dove la confidenzialità del segreto non è garantita(applicazioni eseguite sul dispositivo dell'utente)

3. *Credenziali password proprietario risorse*: utilizzate con applicazioni attendibili, come quelle di proprietà del servizio stesso. Massima credenzialità all'App dandogli le password di accesso (situazioni in cui sono sia client, colui che inserisce le info, sia colui che le gestisce)

4. *Credenziali client*: utilizzate con l'accesso all'API delle applicazioni

Il tipo di concessione del codice di autorizzazione è il più comunemente utilizzato perché è ottimizzato per le applicazioni lato server, dove il codice sorgente non è esposto pubblicamente e la riservatezza del client secret può essere gestita.

Questo è un flusso basato sul reindirizzamento, il che significa che l'applicazione deve essere in grado di interagire con lo user-agent (ovvero il browser Web dell'utente) e ricevere i codici di autorizzazione API che vengono instradati tramite l'user-agent.

Registrazione dell'App

Prima di utilizzare OAuth con la tua applicazione, devi registrare la tua domanda con il servizio (service provider), tramite un modulo di registrazione nella sezione "sviluppatore" o "API" del sito web del servizio, dove fornirai le seguenti informazioni (e probabilmente dettagli sulla tua domanda): Nome dell'applicazione, URL di reindirizzamento del sito Web dell'applicazione o URL di richiamata.

L'URI di reindirizzamento è il punto in cui il servizio reindirizza l'utente dopo aver autorizzato (o rifiutato) l'applicazione, e quindi

la parte della tua applicazione che gestirà i codici di autorizzazione o token di accesso.

Una volta che la tua domanda è stata registrata, il servizio emetterà "credenziali client" sotto forma di identificativo cliente e client secret.
L'ID client è una stringa esposta pubblicamente che viene utilizzata dall'API del servizio per identificare l'applicazione e viene anche utilizzata per creare URL di autorizzazione che vengono presentati agli utenti.
Il client secret viene utilizzato per autenticare l'identità dell'applicazione nell'API del servizio quando l'applicazione richiede di accedere all'account di un utente e deve essere mantenuta privata tra l'applicazione e l'API.

Due tipi di accesso:

- *User authentication:* username e password per accedere a contenuti privati
- *Application only:* voglio accedere ad App pubbliche, non sono necessari username e password:
  - Concateno chiave client + client secret e codifico a 64 bit
  - Otteniamo il token: gli ho detto che App sono e per ottenere il token devo fare una curl -X POST dove gli do info su di me e sulla modalità che uso.

  - Mi dà le Api accessibili con il token precedentemente ottenuto (!curl --request GET\)
    - (Es. un conto è accedere a Facebook come un utente per ottenere info e dati privati personali e su amici, un conto è non accedere e ottenere solo alcune info. Noi nella nostra App accediamo come user)

WebSocket vs SocketIO (F)

Message Queue

Le message queue sono delle code di messaggi che mi permettono di inviare messaggi tra C e S. Non è necessario che entrambe le entità siano attive allo stesso tempo, non perdo messaggi grazie al buffer (evento asincrono).

Un produttore invia messaggi e un consumatore li preleva dal buffer. Se Vc==Vp il buffer non servirebbe poiché sarebbe sincrona, se Vc!=Vp serve un buffer per conservare i messaggi e non perderli, compensando le differenze di velocità e quindi l'asincronia.

Ci sono tantissime tecnologie che forniscono broker e librerie per i protocolli di MQ (come AMQP e MQTT) tra cui quella che approfondiremo è RabbitMQ

AMPQ

Ci sono 6 template di utilizzo di AMQP

1. "HELLO WORLD": un produttore, una coda e un consumatore. P invia i messaggi alla coda e C quando vuole li consuma.

Sender

```
amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    var q = 'hello';

    ch.assertQueue(q, {durable: false});
    // Note: on Node 6 Buffer.from(msg) should be used
    ch.sendToQueue(q, new Buffer('Hello World!'));
    console.log(" [x] Sent 'Hello World!'");
  });
});
```

Receiver

```
var amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    var q = 'hello';

    ch.assertQueue(q, {durable: false});
    console.log(" [*] Waiting for messages in %s. To exit press CTRL+C", q);
    ch.consume(q, function(msg) {
      console.log(" [x] Received %s", msg.content.toString());
    }, {noAck: true});
  });
});
```

2. “WORK QUEUES”: un produttore, una coda e più consumatori. Il msg viene consumato da uno solo dei C.

   L’idea alla base è quella di poter far svolgere un lavoro dividendo i compiti tra più C.
   Uno dei vantaggi dell'utilizzo di una coda di attività è la possibilità di parallelizzare facilmente il lavoro.
   Se stiamo accumulando un arretrato di lavoro, possiamo semplicemente aggiungere più lavoratori e in questo modo, scalare facilmente.I msg verranno distribuiti di default con RR.

   Se un lavoratore muore, perderemo il messaggio che stava solo elaborando e anche tutti i messaggi inviati a questo particolare lavoratore ma non ancora gestiti. Ma non vogliamo, vorremmo che l'attività fosse consegnata a un altro lavoratore. Per essere sicuro che un messaggio non venga mai perso, RabbitMQ supporta i riconoscimenti dei messaggi.

   Un Ack viene inviato dal C per comunicare a RabbitMQ che un particolare messaggio è stato ricevuto, elaborato e che RabbitMQ è libero di cancellarlo. (Aggiungiamo nel C: noAck:false).

   Abbiamo imparato come assicurarci che anche se il consumatore muore, il compito non è perso. Ma i nostri compiti andranno persi se il server RabbitMQ si ferma, abbiamo bisogno di contrassegnare sia la coda che i messaggi come durevoli. (durable:true).

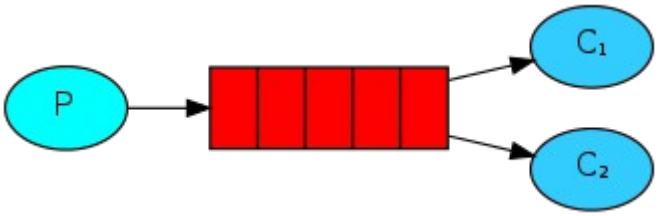
   Questo cambiamento di opzione durevole deve essere applicato sia al codice P che a quello del C. Se un consumatore muore (il suo canale è chiuso, la connessione è chiusa, o la connessione TCP è persa) senza inviare un ack, RabbitMQ capirà che un messaggio non è stato elaborato completamente e lo ri-accoderà.

   Ora dobbiamo contrassegnare i nostri messaggi come persistenti (persistente:true). Potrebbero però accadere situazioni, ad esempio, con due lavoratori, dove tutti i messaggi dispari sono pesanti, un lavoratore sarà costantemente occupato e l'altro non farà praticamente alcun lavoro.

Bene, RabbitMQ non ne sa nulla e continuerà a inviare messaggi in modo uniforme. Ciò accade perché RabbitMQ invia un messaggio solo quando il messaggio entra in coda.

Non considera il numero di messaggi non riconosciuti per un consumatore. Invia solo ciecamente ogni messaggio n-esimo al n-esimo consumatore. Per poterlo sconfiggere, possiamo usare il metodo prefetch con il valore di 1.

Questo dice a RabbitMQ di non dare più di un messaggio a un lavoratore alla volta. In altre parole, non inviare un nuovo messaggio a un lavoratore finché non ha elaborato e riconosciuto quello precedente. Invece, lo invierà al lavoratore successivo che non è ancora occupato.



Sender

```
var q = 'task_queue';
var msg = process.argv.slice(2).join(' ') || "Hello World!";

ch.assertQueue(q, {durable: true});
ch.sendToQueue(q, new Buffer(msg), {persistent: true});
console.log(" [x] Sent '%s'", msg);
```

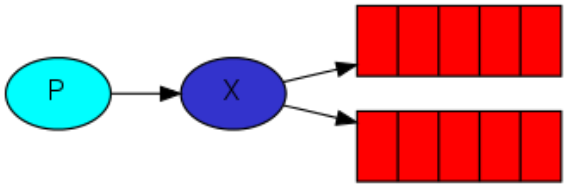
Receiver

```
var q = 'task_queue';

ch.consume(q, function(msg) {
  var secs = msg.content.toString().split('.').length - 1;

  console.log(" [x] Received %s", msg.content.toString());
  setTimeout(function() {
    console.log(" [x] Done");
  }, secs * 1000);
}, {noAck: true});
```

3. “PUBLISH/SUBSCRIBE”: Il produttore produce contenuti a tutti coloro che possono accedere a quei contenuti (possono accedere/sono sottoscrittori di una coda).

   X: exchange, responsabile di smistare msg alle code con una certa politica (in questo caso è quella di inviare msg a tutti i C che insistono su quell’X: fanout).

   Il P conosce solo l’X e non tutti i consumatori poiché lui pubblica solo sull’X.Se arriva un nuovo C, creo una nuova coda e la attacco all’X (binding). X invia msg a tutte le code con cui è stato fatto il binding.



Sender

```
var amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    var ex = 'logs';
    var msg = process.argv.slice(2).join(' ') || 'Hello World!';

    ch.assertExchange(ex, 'fanout', {durable: false});
    ch.publish(ex, '', new Buffer(msg));
    console.log(" [x] Sent %s", msg);
  });

  setTimeout(function() { conn.close(); process.exit(0) }, 500);
});
```

Receiver

```
var amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    var ex = 'logs';

    ch.assertExchange(ex, 'fanout', {durable: false});

    ch.assertQueue('', {exclusive: true}, function(err, q) {
      console.log(" [*] Waiting for messages in %s. To exit press CTRL+C", q.queue);
      ch.bindQueue(q.queue, ex, '');

      ch.consume(q.queue, function(msg) {
        if(msg.content) {
        console.log(" [x] %s", msg.content.toString());
    }
      }, {noAck: true});
    });
  });
});
```

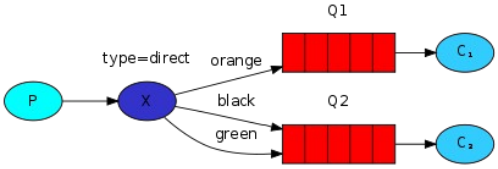
4. “ROUTING”: Il sistema precedente trasmette tutti i messaggi a tutti i consumatori.

   Vogliamo estenderlo per consentire il filtraggio dei messaggi in base alla loro gravità. Ad esempio, potremmo volere lo script che sta scrivendo i messaggi di log sul disco per ricevere solo errori critici e non sprecare spazio su disco nei messaggi di avviso o nel registro delle informazioni.

   Prima utilizzavamo uno scambio di tipo fanout, ora utilizzeremo invece uno scambio diretto. Posso sempre attaccare una nuova coda all’X, ma ora specifico una chiave di routing.

   L'algoritmo di routing dietro uno scambio diretto è semplice: l’X invia un messaggio alle code la cui chiave di routing corrisponde esattamente alla chiave di routing del messaggio.

   La sottoscrizione ad una medesima chiave di routing/attributo può essere fatta da più code, non solo da una; in quel caso il msg verrà recapitato a tutte le code con quella chiave.



Sender

```
var amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    var ex = 'direct_logs';
    var args = process.argv.slice(2);
    var msg = args.slice(1).join(' ') || 'Hello World!';
    var severity = (args.length > 0) ? args[0] : 'info';

    ch.assertExchange(ex, 'direct', {durable: false});
    ch.publish(ex, severity, new Buffer(msg));
    console.log(" [x] Sent %s: '%s'", severity, msg);
  });

  setTimeout(function() { conn.close(); process.exit(0) }, 500);
});
```

Receiver

```
var amqp = require('amqplib/callback_api');

var args = process.argv.slice(2);

if (args.length == 0) {
  console.log("Usage: receive_logs_direct.js [info] [warning] [error]");
  process.exit(1);
}

amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    var ex = 'direct_logs';

    ch.assertExchange(ex, 'direct', {durable: false});

    ch.assertQueue('', {exclusive: true}, function(err, q) {
      console.log(' [*] Waiting for logs. To exit press CTRL+C');

      args.forEach(function(severity) {
        ch.bindQueue(q.queue, ex, severity);
      });

      ch.consume(q.queue, function(msg) {
        console.log(" [x] %s: '%s'", msg.fields.routingKey, msg.content.toString());
      }, {noAck: true});
    });
  });
});
```

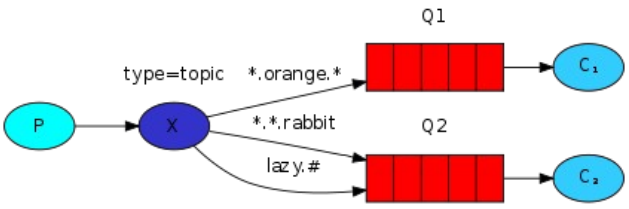
5. “TOPICS”: il sistema precedente presenta ancora delle limitazioni: non può effettuare il routing in base a più criteri.

   I messaggi inviati a uno scambio di argomenti non possono avere un routing_key arbitrario, deve essere un elenco di parole, delimitato da punti.

   Le parole possono essere qualsiasi cosa, ma in genere specificano alcune funzionalità collegate al messaggio. Per esempio, invieremo messaggi che descrivono tutti gli animali. I messaggi verranno inviati con un codice di instradamento composto da tre parole <velocità>. <Colore>. <Specie>. Abbiamo creato tre associazioni: Q1 è associato con chiave di associazione "* .orange. *" E Q2 con "*. *. Rabbit" e "pigro. #". Q1 è interessato a tutti gli animali arancioni. Q2 vuole sapere tutto sui conigli e tutto sugli animali pigri.

   Un messaggio con una chiave di instradamento impostata su "quick.orange.rabbit" verrà consegnato a entrambe le code. Anche il messaggio "lazy.orange.elephant" andrà ad entrambi. D'altra parte "quick.orange.fox" andrà solo alla prima coda e "lazy.brown.fox" solo al secondo. "lazy.pink.rabbit" verrà consegnato alla seconda coda solo una volta, anche se corrisponde a due associazioni. "quick.brown.fox" non corrisponde ad alcuna associazione quindi verrà scartata.

   Se rompiamo il nostro contratto e inviamo un messaggio con una o quattro parole, come "orange" o "quick.orange.male.rabbit" questi messaggi non corrisponderanno a nessun binding e andranno persi.



Sender

```
var amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    var ex = 'topic_logs';
    var args = process.argv.slice(2);
    var key = (args.length > 0) ? args[0] : 'anonymous.info';
    var msg = args.slice(1).join(' ') || 'Hello World!';

    ch.assertExchange(ex, 'topic', {durable: false});
    ch.publish(ex, key, new Buffer(msg));
    console.log(" [x] Sent %s:'%s'", key, msg);
  });

  setTimeout(function() { conn.close(); process.exit(0) }, 500);
});
```

Receiver

```
var amqp = require('amqplib/callback_api');

var args = process.argv.slice(2);

if (args.length == 0) {
  console.log("Usage: receive_logs_topic.js <facility>.<severity>");
  process.exit(1);
}

amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    var ex = 'topic_logs';

    ch.assertExchange(ex, 'topic', {durable: false});

    ch.assertQueue('', {exclusive: true}, function(err, q) {
      console.log(' [*] Waiting for logs. To exit press CTRL+C');

      args.forEach(function(key) {
        ch.bindQueue(q.queue, ex, key);
      });

      ch.consume(q.queue, function(msg) {
        console.log(" [x] %s:'%s'", msg.fields.routingKey, msg.content.toString());
      }, {noAck: true});
    });
  });
});
```

6. “RPC”: Se abbiamo bisogno di eseguire una funzione su un computer remoto e attendere il risultato useremo RabbitMQ per costruire un sistema RPC: un client e un server RPC scalabile.

   C invia delle richieste su una coda al S che risponderà sulla coda di callback, unica per ogni client. Ciò solleva un nuovo problema, avendo ricevuto una risposta in tale coda non è chiaro a quale richiesta appartienga la risposta.

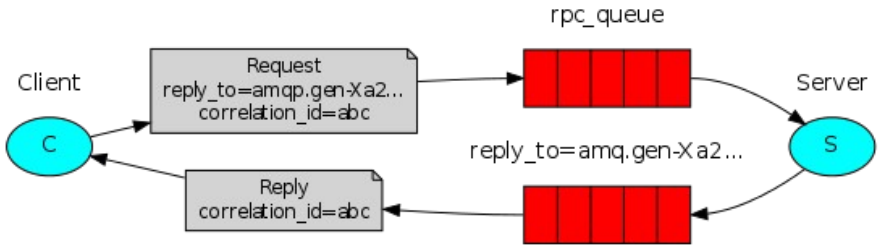
   Useremo quindi il correlation_id. Lo imposteremo su un valore univoco per ogni richiesta. Successivamente, quando riceviamo un messaggio nella coda di callback, esamineremo il suo valore e, in base a ciò, saremo in grado di abbinare una risposta a una richiesta.

   Se vediamo un valore correlation_id sconosciuto, possiamo tranquillamente scartare il messaggio - non appartiene alle nostre richieste.

   Il nostro RPC funzionerà in questo modo: All'avvio del client, viene creata una coda di richiamata esclusiva anonima. Per una richiesta RPC, il client invia un messaggio con due proprietà: reply_to, che è impostato sulla coda di callback e correlation_id, che è impostato su un valore univoco per ogni richiesta.

   La richiesta viene inviata a una coda rpc_queue. L'operatore RPC (server) è in attesa di richieste su tale coda. Quando viene visualizzata una richiesta, esegue il lavoro e invia un messaggio con il risultato alla C, utilizzando la coda dal campo reply_to. Il client attende i dati sulla coda di callback.

   Quando viene visualizzato un messaggio, controlla la proprietà correlation_id. Se corrisponde al valore della richiesta, restituisce la risposta all'applicazione.



Sender

```
var amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    var q = 'rpc_queue';

    ch.assertQueue(q, {durable: false});
    ch.prefetch(1);
    console.log(' [x] Awaiting RPC requests');
    ch.consume(q, function reply(msg) {
      var n = parseInt(msg.content.toString());

      console.log(" [.] fib(%d)", n);

      var r = fibonacci(n);

      ch.sendToQueue(msg.properties.replyTo,
        new Buffer(r.toString()),
        {correlationId: msg.properties.correlationId});

      ch.ack(msg);
    });
  });
});

function fibonacci(n) {
  if (n == 0 || n == 1)
    return n;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Receiver

```
var amqp = require('amqplib/callback_api');

var args = process.argv.slice(2);

if (args.length == 0) {
  console.log("Usage: rpc_client.js num");
  process.exit(1);
}

amqp.connect('amqp://localhost', function(err, conn) {
  conn.createChannel(function(err, ch) {
    ch.assertQueue('', {exclusive: true}, function(err, q) {
      var corr = generateUuid();
      var num = parseInt(args[0]);

      console.log(' [x] Requesting fib(%d)', num);

      ch.consume(q.queue, function(msg) {
        if (msg.properties.correlationId == corr) {
          console.log(' [.] Got %s', msg.content.toString());
          setTimeout(function() { conn.close(); process.exit(0) }, 500);
        }
      }, {noAck: true});

      ch.sendToQueue('rpc_queue',
      new Buffer(num.toString()),
      { correlationId: corr, replyTo: q.queue });
    });
  });
});

function generateUuid() {
  return Math.random().toString() +
         Math.random().toString() +
         Math.random().toString();
}
```

MQTT

Versione “light” di AMQP ma più “limitata”: tutti i paradigmi visti in AMQP ti danno una sorta di potere espressivo, MQTT lavora solo

sul publish/subscribe con i topic (che consente quindi un’organizzazione gerarchica).

- Basato su TCP
- Asincrono
- Publish/Subscribe
- Payload agnostic: dentro il payload posso mettere qualsiasi cosa
- Few verbs: posso fare poche cose ma fatte molto bene

In MQTT tutti sono sia Publisher che subscriber, possono quindi sia inviare sia ricevere messaggi, inviandoli al broker che, in base al topic del msg, selezionerà i destinatari.

In questo caso avrò un unico spazio di concetti (albero) quindi impiego O(logn) di ricerca degli attributi (es se voglio sapere tutte le info sulla room1 scriverò building1/floor1/room1/*).
Qualità del servizio (QoS): at most once, at least once, exactly once.

WebSocket vs SocketIO

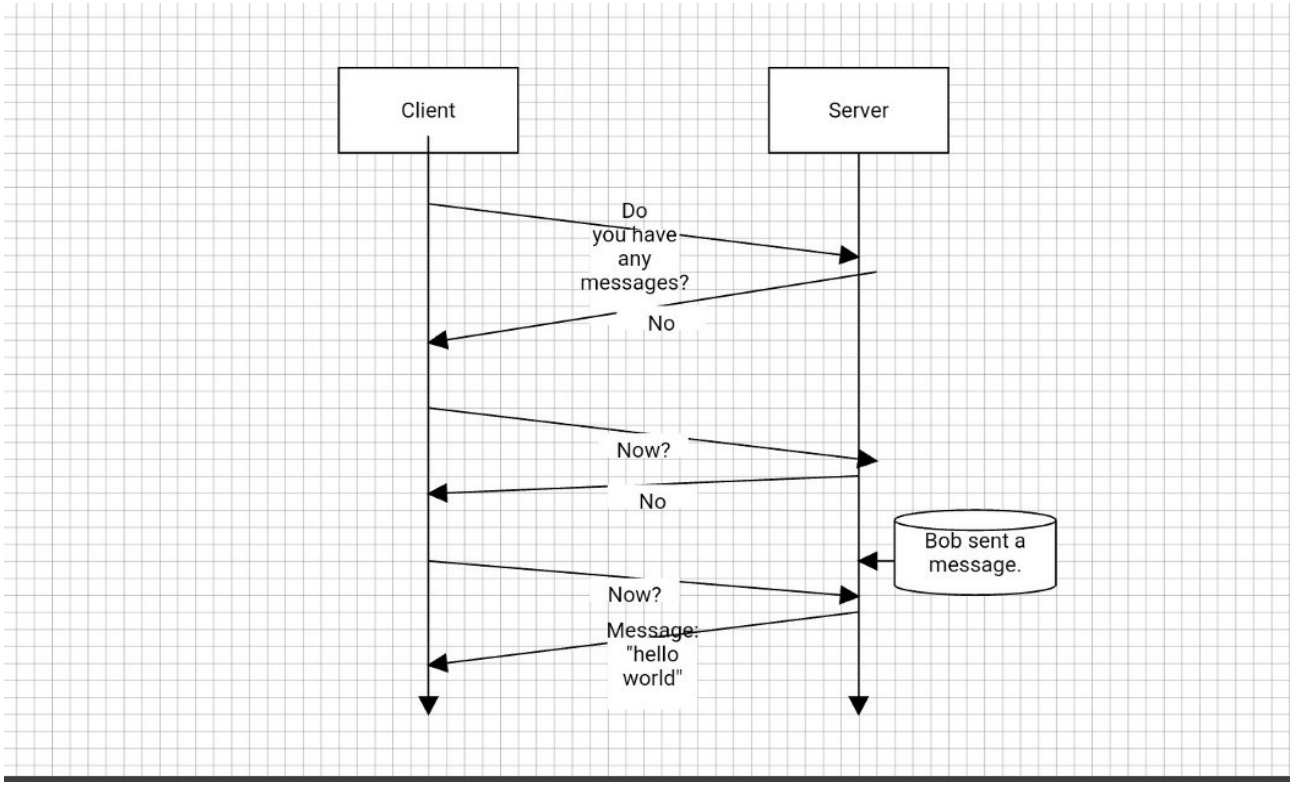
WebSocket fornisce un'alternativa gradita alle tecnologie AJAX.

WebSocket è una tecnologia web che fornisce possibilità di comunicazione asincrona e full duplex attraverso una singola connessione TCP.

L’aspetto cruciale e fondamentale della nascita e dell’utilizzo delle WebSocket risiede nel fatto che in determinate situazioni l’aspetto del “real-time” risulta essere primario.

Prima dell’avvento delle WebSocket una possibile soluzione per implementare applicazioni quasi real-time era quello di sviluppare una sorta di Polling. (Ad esempio supponiamo di dover realizzare una sorta di notifica in una applicazione web; il client ogni tot secondi richiede al server se ci sono nuovi messaggi e in caso affermativo mostra il nuovo messaggio.)

POLLING



Questo meccanismo è molto inefficiente:

- Il dato non viene ricevuto in real-time.
- Spreco immenso di banda.
- Overhead del server: il server si troverà costantemente bombardato dai client connessi.

Funzionamento di ws

Ogni connessione WebSocket ha inizio con una richiesta HTTP inviata dal client al server e chiamata “WebSocket handshake”.

Questa richiesta è una normale richiesta HTTP, salvo l’inclusione di un ulteriore riga di intestazione “Upgrade: websocket” che indica che il client vuole aggiornare la connessione ad un protocollo diverso, in questo caso a WebSocket.

Se il server accetta l’handshake e supporta il protocollo, accetta l’aggiornamento e lo comunica tramite l’header *Upgrade.*
A questo punto l’iniziale connessione HTTP viene sostituita con una connessione WebSocket che fa uso del medesimo protocollo TCP/IP.
Con gli WebSocket puoi trasferire dati senza incorrere nell’overhead associato alle tradizionali richieste HTTP. L’utilizzo di questo sistema di messaggistica basato su frame consente di

ridurre la quantità di dati non a carico utile trasferiti, con conseguente riduzione significativa della latenza.

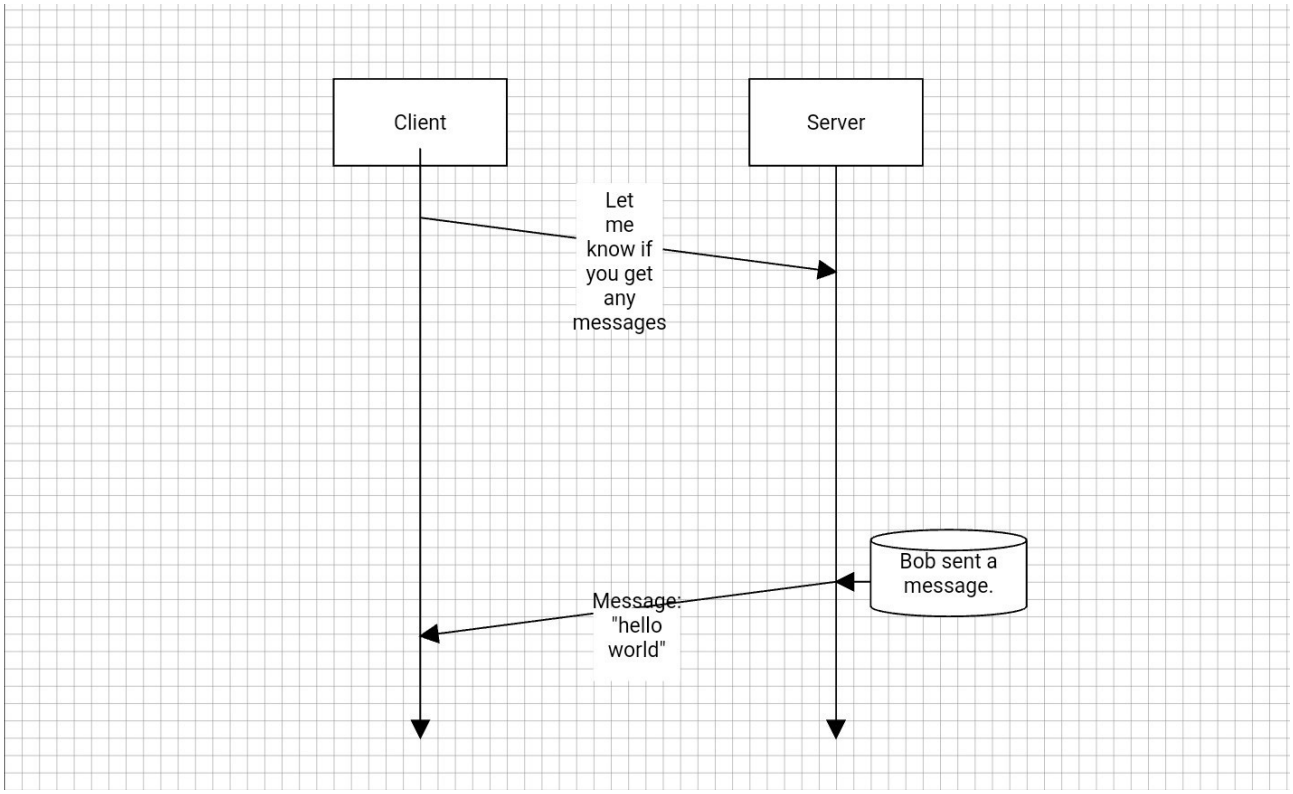
Vantaggi ws

- *Vero real-time* poiché l'unico delay che rimane è quello del trasferimento dati.

- *Meno overhead lato server* grazie al fatto che se non ci sono nuovi messaggi il server non viene stressato da inutili richieste (le risposte da parte del server sono event-driven).

- *Bassa latenza e nessuno spreco di banda* considerare che per stabilire una normale connessione HTTP devi prima stabilire una connessione TCP, quindi inviare una richiesta GET con un header piuttosto grande e infine ricevere la risposta del server (insieme a un'altra grande header). Con un WebSocket aperto si riceve invece semplicemente la risposta (nessuna richiesta necessaria) fornita con un header decisamente più piccolo: da **due** byte per i frame piccoli, fino a 10 byte per i frame ridicolmente grandi.

WebSocket



Socket IO

SocketIO è un modulo di Node.js che consente la comunicazione in tempo reale, bidirezionale e basata su eventi tra browser e server.

Questo tipo di tecnologia viene utilizzata per creare applicazioni quali i sistemi di comunicazione in tempo reale, i giochi multiutente, le applicazioni di messaggistica e così via.

SocketIO **NON** è un'implementazione WebSocket. Sebbene SocketIO utilizzi effettivamente WebSocket come trasporto quando possibile, aggiunge alcuni metadati a ciascun pacchetto: il tipo di pacchetto, lo spazio dei nomi e l'ID ack quando è necessario un riconoscimento del messaggio. Ecco perché un client WebSocket non sarà in grado di connettersi correttamente ad un server SocketIO e viceversa.

Inoltre se non diversamente specificato, un client disconnesso proverà a riconnettersi dopo la perdita della connessione, finché il server non sarà nuovamente disponibile.

Uno dei punti di forza di SocketIO è l’affidabilità. A tale scopo si affida a EngineIO (libreria a livello inferiore rispetto a SocketIO), che prima stabilisce prima una connessione a polling lungo, quindi tenta di eseguire l'aggiornamento a trasporti migliori che sono "testati" sul lato, come WebSocket.

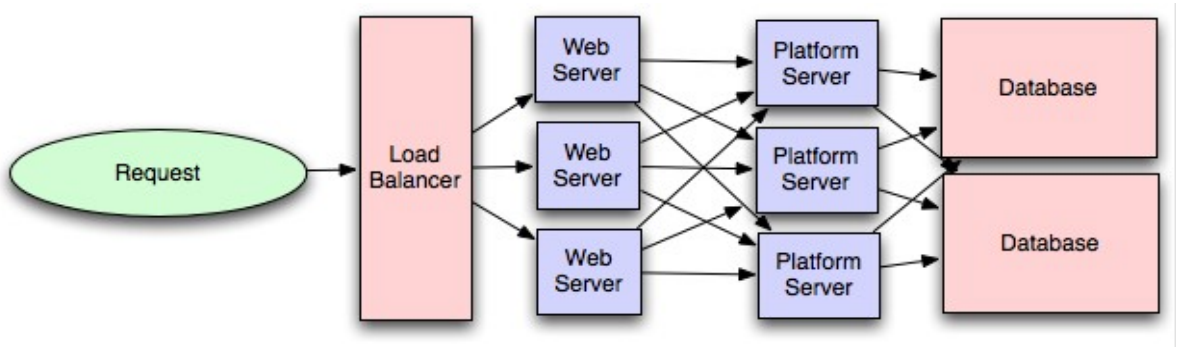
Internet of Things (IoT)

Negli ultimi anni, la miniaturizzazione e la riduzione dei costi HW (legge di Moore) e la capacità di supportare milioni o addirittura miliardi di dispositivi connessi hanno spinto l'Internet of things (IoT).

IoT è la rete di dispositivi fisici, veicoli, elettrodomestici e altri elementi incorporati con elettronica, software, sensori, attuatori e connettività di rete che consentono a questi oggetti di connettersi e scambiarsi dati.

Ogni cosa è identificabile in modo univoco attraverso il suo sistema informatico integrato, ma è in grado di interagire all'interno dell'infrastruttura Internet esistente.

DOMANDE GENERALI



1) Che cos’è il Load Balancing?

Load Balancing vuol dire bilanciamento di carico e consiste nel distribuire il carico di uno specifico servizio tra più server.

L’obiettivo del Load balancing è di ottimizzare l’uso delle risorse a disposizione, massimizzandone la capacità produttiva e minimizzando il tempo di risposta, ed evitando il sovraccarico della singola risorsa. Usando il load balancing con molteplici risorse invece di un singolo componente, l’affidabilità e la scalabilità dell’architettura vengono notevolmente incrementate.

2) Quali sono i vantaggi che Load Balancing offre?

Tra i primi vantaggi del load balancer c’è sicuramente quello di avere tempi di caricamento e di risposta più brevi.

La sicurezza della stabilità del server è un altro dei vantaggi, dal momento che il traffico di un server lento viene automaticamente inoltrato ad un altro server. Inoltre, nel momento il cui un server non è raggiungibile, con il load balancer si ha la garanzia che il sito rimanga disponibile in quanto ci sarà almeno un altro server raggiungibile.

3) Che cos’è un Web Server?

Un Web Server non è altro che un’applicazione software che, in esecuzione su un server, è in grado di gestire le richieste di trasferimento di pagine web di un client, ovverosia di un browser. In particolare un Web Server si occupa di pagine statiche, file prelevati dal server e visualizzati sul browser.

Esempi di Web server sono Nginx e Apache, il primo è un Web Server a eventi, il secondo a thread.

Il problema del Web Server sta nel fatto che se la mia richiesta è di tipo “POST” avrò bisogno di un Application Server, perché in base al metodo avrò non avrò un risultato atteso stabilito a priori.

4) Che cos’è un Application Server/Platform Server?

Un Application Server è un server che gestiste pagine dinamiche, cioè pagine in cui il contenuto non è deciso a priori ma può variare in base a condizioni di vario genere.

Per Esempio quando su un motore di ricerca effettuate una interrogazione, la pagina dei risultati che vi viene presentata non è già esistente prima della vostra richiesta. Solo dopo che il server ha eseguito la sua ricerca sulla parola chiave da voi indicata viene creato il codice per presentarvi i risultati. Un esempio di Application Server è Node.js che utilizza il linguaggio Javascript per gestire le varie richieste che gli vengono fatte.

5) Che cos’è un Database?

Un database (abbreviato in*DB*) è un'entità nella quale è possibile immagazzinare dei dati in modo strutturato e con la minima ripetizione possibile. Questi dati devono poter essere utilizzati dai programmi e da utenti differenti.
L’accesso ai dati del dati è una semplificazione di un framework di nome REST. Quindi invece di utilizzare una libreria utilizzo HTTP per utilizzare REST e quindi CRUD per gestire le informaizioni all’interno del Database.