Alma Mater Studiorum - Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA

in

TECNOLOGIE WEB T

Sviluppo e Analisi delle Prestazioni di Applicazioni Web Nuxt-based in Cloud AWS

Candidato:
Valerio Iacobucci

Relatore:
Chiar.mo Prof. Ing.
Paolo Bellavista

Anno Accademico 2023/2024

Indice

Introduzione:

Questo lavoro di tesi focalizza l'attenzione sul framework per la realizzazione di applicazioni Web Nuxt e sulla libreria per *Object relational mapping* TypeORM, proponendo alcune soluzioni implementative intese a migliorare la qualità, la manutenibilità e la scalabilità del codice.

Obiettivi:

L'obiettivo principale è quello di fornire un'analisi critica delle soluzioni di design proposte, fornendo valutazioni di performance e di sicurezza in base a test effettuati su applicazioni di esempio. Il deploy di queste applicazioni è effettuato su un'infrastruttura *Amazon Web Services* (AWS), piattaforma sulla quale ho lavorato durante il Tirocinio curriculare.

Capitolo 1

Linee evolutive

Il Web è la piattaforma software più estesa al mondo e la sua evoluzione è stata guidata da una serie di innovazioni tecnologiche che hanno permesso di realizzare applicazioni sempre più complesse e performanti. Questo capitolo ripercorre brevemente le linee evolutive del Web, partendo dalle pagine statiche fino ad arrivare alle Applicazioni Web moderne.

1.1 Pagine statiche

Il primo modello del World Wide Web era orientato a facilitare la condivisione di documenti, permettendo ai lettori di esplorarli attraverso collegamenti tra le pagine. Il WWW consisteva in una combinazione di applicazioni, di protocolli e di linguaggi di marcatura progettati e rilasciati presso il CERN, principalmente ad opera di Tim Berners Lee e Robert Calliau, a ridosso degli anni '90:

Browser e Server: Con un browser Web installato nel proprio sistema informatico, un utente può visualizzare pagine Web e scegliere di seguire i collegamenti ipertestuali per accedere ad altre pagine. Il server Web è responsabile di fornire le pagine Web richieste dai client, come i browser.

Protocollo HTTP: È il protocollo di comunicazione di livello applicativo (OSI 7) che permette la trasmissione di informazioni tra client e server Web. Un browser contatta un server Web inviando una richiesta HTTP ad un determinato URL e il server risponde con una risposta HTTP contenente i dati richiesti. HTTP è un protocollo stateless, non mantiene informazioni sullo stato della comunicazione, quindi sta all'applicazione gestire eventuali sessioni o autenticazioni.

Linguaggio HTML: (HyperText Markup Language) è un linguaggio di marcatura utilizzato per la realizzazione di pagine Web. HTML definisce la struttura e il contenuto di una pagina Web attraverso l'uso di tag e attributi e consente di incorporare elementi multimediali. Il browser Web interpreta il codice HTML e mostra la pagina Web all'utente.

Nel 1993, il primo browser Web grafico, Mosaic, introdusse il supporto per le immagini, per i form (dei moduli compilabili dall'utente con opzioni) e per i collegamenti ipertestuali, e successivamente Netscape Navigator 1.0 introdusse il supporto per CSS, il linguaggio che permise da subito agli sviluppatori di personalizzare la resa grafica della loro pagina. Queste innovazioni contribuirono a rendere il Web più accessibile e visivamente attraente per un pubblico maggiore.

1.2 Pagine dinamiche

I primi browser Web erano in grado di visualizzare solo pagine statiche, il che significa che il contenuto di una determinata pagina non cambiava in base all'interazione dell'utente[1].

Le prime realizzazioni di pagine dinamiche furono rese possibili grazie agli hyperlinks ed ai form, con i quali i browser potevano effettuare richieste al server inviando i parametri forniti dall'utente. Il server poteva elaborare tali richieste e restituire una *nuova pagina HTML* in base ai parametri ricevuti. Questo processo era gestito da programmi detti **CGI** (Common Gateway Interface), da eseguire sul server per generare pagine dinamicamente, cioè al momento della richiesta.

I linguaggi di programmazione utilizzati all'epoca per scrivere programmi CGI erano principalmente:

Linguaggi di basso livello o di scripting

- Linguaggi di basso livello come C o C++, sono performanti ma di difficile progettazione e manutenzione.
- Linguaggi di scripting come Perl o Shell UNIX, erano più facili da utilizzare ma meno efficienti: comodi per la manipolazione di stringhe e file, ma non per la gestione di strutture dati complesse.

Linguaggi di templating Successivamente, dal 1995 in poi, emersero alcuni Linguaggi di templating che consentono di incorporare codice dinamico all'interno di pagine HTML dal lato server.

- Java Server Pages (JSP) è un'estensione di Java, quindi era possibile usare tutte le librerie di questo linguaggio molto popolare all'epoca.
- PHP è un linguaggio interpretato che ha avuto molto successo per oltre un decennio[2] grazie alla sua facilità d'uso.

```
1  <?php
2  $username = $_POST["username"];
3  $password = $_POST["password"];
4
5  $query = sprintf( "SELECT * FROM Users WHERE username='%s' AND password='%s'",
6      $username, $password);
7  $result = mysql_query($query, $conn);
8
9  if (mysql_num_rows($result) > 0){
10 ?>
11     <h1>Benvenuto <?php echo $username; ?></h1>
12     ...
13 <?php
14     } else {
```

[1] Gli unici effetti che si potevano apprezzare immediatamente dopo un'interazione dell'utente erano quelli di CSS, ad esempio il cambio di colore di un link al passaggio del mouse.

[2] Indice TIOBE per PHP, si può vedere come il suo utilizzo sia diminuito a partire dal 2010.

```
    ?>
        <h1>Accesso negato</h1>
        <p>Torna alla <a href="login">pagina di login</a></p>
    <?php
    }
?>
```

Un esempio di pagina di autenticazione in PHP, che riflette lo stile di programmazione tipico dell'epoca[3]. È da notare come il codice HTML da inviare al browser sia inserito direttamente all'interno del codice da mantenere privato nel lato server, rendendone difficile la manutenzione per via della *confusione tra logica di presentazione e logica di business*. Vengono poi adoperati 3 linguaggi diversi (PHP, HTML, SQL), soluzione non ottimale per la leggibilità, che si aggiunge ai problemi di sicurezza legati all'*interpolazione* di stringhe all'interno di query SQL.

1.3 Pagine attive con Javascript

Il dinamismo delle pagine Web supportato da server CGI e linguaggi di scripting era comunque limitato per via del caricamento di nuove pagine ad ogni richiesta. Non era possibile aggiornare parzialmente la pagina, ma solo scaricarne una nuova. Nel 1995 il Netscape Navigator 2.0 introdusse il supporto ad un nuovo linguaggio di scripting, che successivamente venne chiamato Javascript, realizzato da Brendan Eich, per ovviare a questo problema.

Gestione di eventi e manipolazione del DOM: Uno script Javascript, distribuito all'interno di una pagina HTML, può essere eseguito dal browser Web in risposta a determinati eventi dell'utente. Inizialmente il motore di esecuzione era sincrono, cioè bloccava l'esecuzione del codice fino al completamento dell'operazione e le possibilità di Javascript si limitavano alla manipolazione a *runtime*[4] del DOM (Document Object Model).

Richieste HTTP asincrone: Le pagine web, erano diventate *attive*, ma tutte le risorse dovevano essere inserite nella pagina inviata con la prima risposta HTTP. Nel 1999 però, il browser Internet Explorer 5 introdusse un'estensione del linguaggio Javascript, che disponeva di un oggetto chiamato *XMLHttpRequest*, in grado effettuare richieste HTTP asincrone al server e dunque ricevere risposte senza dover ricaricare l'intera pagina. Così si gettavano le basi per la realizzazione di *Single Page Applications*.

La libreria jQuery, rilasciata nel 2006, ha rivestito una particolare importanza perché semplificava la manipolazione del DOM e le richieste HTTP, fornendo un'interfaccia più semplice e omogenea rispetto ai diversi browser, che esponevano API diverse e non ancora standardizzate.

```
$("#update").click(function () {
    $.ajax({
        // scaricamento asincrono
        url: "data.json",
        type: "GET",
        dataType: "json",
```

[3] La libreria mysql di Micheal Widenius per PHP 2 risale al 1996.

[4] Il momento in cui la pagina è resa attiva con Javascript.

```
        success: function (data) {
            var content = "";
            for (var i = 0; i < data.length; i++) {
                content += "<p>" + data[i].name + "</p>";
            }
            $("#content").html(content);
        },
    });
});
```

Nel frammento di codice jQuery, è messo in evidenza uno stile *imperativo* di definizione del comportamento dell'interfaccia utente, poco manutenibile per applicazioni complesse.

1.4 Node.js e Javascript lato server

La standardizzazione di Javascript procedette attraverso le varie versioni di ECMAScript che definivano le nuove funzionalità del linguaggio e, di conseguenza, dei browser Web. Nel 2008 fu rilasciata la prima versione di Google Chrome, un browser che, oltre ad includere caratteristiche appetibili per gli utenti finali, disponeva del motore di esecuzione V8 per Javascript. Questo *engine* apportò dei sostanziali miglioramenti di prestazioni[5] rispetto alla competizione e venne rilasciato come *Open source software*.

Nel 2009, Ryan Dahl iniziò a lavorare, basandosi sul codice di V8, a Node.js, un interprete di Javascript in modalità headless[6], al quale aggiunse la capacità di accedere al filesystem, di esporre servizi HTTP e di accettare connessioni in maniera non bloccante.

In questo modo si poterono realizzare non solo applicazioni **frontend** ma anche **backend** con Javascript, abilitando sempre più novizi alla creazione di siti Web completi. Attorno a Node crebbe una comunità di sviluppatori che contribuirono, secondo i principi dell'Open source, alla creazione di un ecosistema di librerie, che potevano essere installate tramite il gestore di pacchetti NPM.

```
const http = require("http");
const mysql = require("mysql");

const connection = mysql.createConnection({
    /* ... */
});

const server = http.createServer((req, res) => {
    if (req.method === "POST" && req.url === "/login") {
        var username, password;
        // unmarshalling della query string ...
        var login = "SELECT * FROM Users WHERE username = ? AND password = ?";
        connection.query(login, [username, password], (err, rows) => {
```

[5] Google Chrome announcement: in questo video si può vedere come l'esecuzione di Javascript su Chrome sia di circa 60 volte più veloce che su Internet Explorer 8.

[6] Cioè senza interfaccia grafica.

```
14              if (rows.length > 0) {
15                  res.writeHead(200, { "Content-Type": "text/html" });
16                  res.end("<h1>Benvenuto " + username + "</h1>");
17              } else {
18                  res.writeHead(401, { "Content-Type": "text/html" });
19                  res.end("<h1>Accesso negato</h1>");
20              }
21          });
22      }
23  });
24  server.listen(80, () => {
25      console.log("Server in ascolto alla porta 8080");
26  });
```

In questo frammento di codice è mostrato l'utilizzo della libreria "http" fornita di default da Node e della libreria "mysql" di Felix Geisendörfer, una delle prime per l'accesso a database da Node. È da notare l'architettura a callback, che permette di gestire in maniera asincrona le richieste HTTP e le query al database.

In questo esempio le query SQL sono parametrizzate, facendo uso di *Prepared statements*, per evitare attacchi di tipo injection, ma rimangono cablate all'interno di stringhe, rendendo il codice vulnerabile a errori di sintassi e di tipo.

1.5 Applicazioni Web orientate a componenti

Anche con Node fu possibile realizzare applicazioni Web *monolitiche*, parimenti al templating PHP, usando librerie come EJS di TJ Holowaychuk.

Tuttavia le tendenze di quel periodo (circa 2010) si discostarono dal modo tradizionale di scrivere applicazioni Web, basate su pagine generate lato server, per passare a un modello di **client-side rendering**. Secondo questo modello il server invia al browser una pagina HTML con un DOM minimo, corredato di script JS che si occupano di popolare a runtime il DOM con contenuti e di gestire le logiche di presentazione.

Le applicazioni renderizzate lato cliente potevano beneficiare di una maggiore *reattività* e di una migliore esperienza utente, essendo basate su una pagina unica che veniva aggiornata in maniera incrementale, aggirando i caricamenti di intere pagine da richiedere al server. Le richieste al server, essendo asincrone, potevano essere gestite in modo meno invasivo rispetto a prima: mentre la comunicazione avveniva in background, l'utente poteva continuare ad interagire con l'applicazione.

Il vantaggio di questo paradigma da parte degli sviluppatori era la possibilità di scrivere la logica di presentazione interamente in Javascript, sfruttando il sistema di oggetti e la modularità del linguaggio in maniera più espressiva rispetto al templating o all'uso imperativo delle API DOM.

L'idea centrale delle nuove tendenze *CSR* era quella di progettare l'interfaccia utente partendo da parti più piccole, chiamate **componenti**, e riutilizzabili all'interno dell'intera applicazione. Lo stile assunto era *dichiarativo*[7]. Ad ogni componente sono associati:

[7] In questo contesto, uno stile dichiarativo è riferito ad un approccio alla programmazione in cui si descrive cosa il programma deve fare piuttosto che come farlo. Con jQuery si dovevano specificare esplicitamente i passaggi per

- un template HTML, più piccolo e gestibile rispetto ad una pagina intera.
- un foglio CSS, per la stilizzazione.
- il codice Javascript che ne definisce la logica di interazione.

Tra gli esempi più noti di sistemi di componenti:

Angular.js Uno dei primi framework a proporre un modello di componenti, sviluppato in Google e rilasciato nel 2010. Angular.js introduceva il concetto di *two-way data binding*, cioè la possibilità di sincronizzare automaticamente i dati tra il modello e la vista.

React.js La libreria di componenti sviluppata da un team interno di Facebook e rilasciata nel 2013. React introduceva il concetto di *Virtual DOM*, una rappresentazione in memoria del DOM reale, che permetteva di calcolare in maniera efficiente le differenze tra due stati del DOM e di applicare solo le modifiche necessarie. Per questi miglioramenti nella performance venne adottato moltissimo[8]. Da React in poi, lo sviluppo di pagine Web ha riguardato un livello più astratto rispetto all'esecuzione del classico codice Javascript che manipola direttamente il DOM. Inteso così, il browser diventa l'interprete di un codice intermedio sul quale gli sviluppatori non mettono mano direttamente.

Vue.js Partito come progetto personale di Evan You e rilasciato nel 2014, Vue si proponeva come un'alternativa più flessibile e meno verbosa rispetto ad Angular e React, dai quali riprende il binding bidirezionale e il Virtual DOM. È la libreria di componenti usata da Nuxt.

```
<script setup>
    import { RouterView } from "vue-router";
    import HelloWorld from "./components/HelloWorld.vue";
</script>

<template>
    <HelloWorld msg="Ciao mondo" />
    <RouterView />
</template>

<style scoped>
    background-color: #f0f0f0;
</style>
```

Un esempio moderno di applicazione Vue 3, che mostra l'utilizzo di un componente "HelloWorld" all'interno di un template principale, e di un RouterView per la navigazione tra le pagine. Questi componenti sono definiti in file distinti, per favorire la separazione delle preoccupazioni, ed importati nel file principale come se fossero moduli Javascript. Ogni volta che compaiono in un template assumono un comportamento dettato dalla

manipolare il DOM, mentre i componenti permettono di definire il comportamento dell'interfaccia utente attraverso delle dichiarazioni più astratte e concise.

[8] Github - React - la più popolare in base numero di stelle su Github.

loro definizione (il loro template) e dal loro *stato* di istanza. Si noti come il componente HelloWorld sia parametrizzato con un attributo `msg`, che ne consente un utilizzo più flessibile.

```
<html lang="en">
    <head>
        <title>Vite App</title>
    </head>
    <body>
        <div id="app"></div>
        <script type="module" src="/src/main.js"></script>
    </body>
</html>
```

Il DOM minimo che viene distribuito da una applicazione Vue 3. La pagina viene assemblata lato client, a partire dal così detto *entry point*: l'elemento con id "app", in cui viene montata l'applicazione. Si noti che nel caso che Javascript non sia abilitato nel client il contenuto dell'applicazione non verrà visualizzato affatto.

1.6 Typescript e ORM

Le basi di codice Javascript iniziarono a diventare sempre più complesse quando anche i team di sviluppatori di grandi compagnie iniziarono ad adottare i framework a componenti. A partire dal 2014 anche applicazioni Web come Instagram, Netflix e Airbnb incorporarono React nei loro stack tecnologici per realizzare interamente l'interfaccia utente.

Javascript, essendo un linguaggio interpretato e debolmente tipizzato, non era in grado di garantire la correttezza del codice e i test di unità erano fatti in modo *behavior driven*, cioè basati sul comportamento dell'applicazione e non sulla tipizzazione dei dati. Questo spesse volte portava ad errori, difficili da individuare e correggere, soprattutto in applicazioni di grandi dimensioni.

Nel 2012 Anders Hejlsberg ed il suo team interno a Microsoft iniziarono a lavorare al linguaggio Typescript, una estensione di Javascript, realizzando un **compilatore** in grado di rilevare errori di tipo in con analisi statica. Typescript permette anche di sfruttare le funzionalità delle nuove versioni di ECMAScript in modo *retrocompatibile*, cioè facendo *transpiling*[9] verso una specifica di ECMA inferiore, per usarle anche sui browser deprecati.

L'adozione di Typescript è stata pressoché immediata, per il motivo che la conversione di basi di codice a partire da Javascript vanilla[10] era a costo zero: ogni sorgente Javascript è valido Typescript. Typescript ha avuto successo non solo lato client, ma anche lato server. Sono comparse infatti alcune librerie di supporto all'accesso a database basate sul pattern **ORM**, *Object-relational mapping*, quindi capaci di mappare il modello dei dati presente nel database a strutture dati proprie di Typescript. Librerie notevoli di questo tipo sono:

[9] Wikipedia - Source-to-source compiler - il transpiling è il processo di traduzione automatica di codice sorgente da un linguaggio ad un altro.

[10] Con "vanilla" ci riferisce a Javascript senza estensioni, quindi al codice che può eseguire nativamente sui browser conformi alle specifiche ECMA. Typescript invece è un *superset*, quindi ha un insieme di espressioni sintattiche più grande ma che comprende interamente quello di Javascript.

Sequelize È stata una delle prime, il progetto è iniziato nel 2010 quindi funzionava con Javascript vanilla, ma si è evoluta fino a supportare le migliorie di Typescript ed una moltitudine di DBMS.

TypeORM Offre supporto a Typescript nativamente. È illustrata con dettaglio nel capitolo 2.

L'evoluzione dei sistemi per fare query a basi di dati da Javascript è poi diramata in direzioni diverse, da quelli che usano protocolli applicativi binari (basati ad esempio su gRPC) a quelli che usano linguaggi di query specifici (come GraphQL), ad ORM che introducono nuovi linguaggi di definizione dei modelli (come Prisma).

1.7 Ritorno al server-side rendering

Dal lancio di React sempre più applicazioni Web hanno fatto uso della tecnica CSR per via della migliorata esperienza utente e di sviluppo. Questo approccio ha portato però una serie di nuovi problemi e limitazioni legate al meccanismo di rendering.

Performance su dispositivi lenti I *bundle* Javascript che vengono generati per le applicazioni CSR sono spesso onerosi in termini di risorse, e la loro esecuzione su dispositivi con capacità di calcolo limitate può risultare lenta e insoddisfacente per l'utente.

Search engine optimization I siti Web che fanno uso di CSR più difficilmente sono indicizzabili dai *crawler* dei motori di ricerca, questo può portare a problemi di esposizione ridotta.

First contentful paint È il tempo che intercorre tra la cattura della risposta HTTP del server e il momento in cui viene visualizzato a schermo dal browser il primo elemento di contenuto significativo per l'utente. Nelle applicazioni CSR questa durata spesso eccede quella massima suggerita da Google[11].

Cumulative layout shift Per il motivo che gli aggiornamenti dell'interfaccia vengono vengono resi graficamente in maniera sequenziale nel browser, potrebbero esserci dei fastidiosi spostamenti di elementi visivi nell'interfaccia durante la fase di caricamento.

Accessibility Per gli stessi motivi che portano al cumulative layout shift, ci potrebbero essere degli impedimenti di accessibilità per chi usa metodi di input alternativi o per gli screen-reader che aiutano le persone non vedenti nella fruizione delle pagine Web.

Per questi motivi, a partire dal 2016, sono emerse delle nuove tendenze che hanno portato ad un ritorno al server-side rendering, in combinazione con i sistemi basati su componenti, per unire i vantaggi di entrambi i modelli. Esempi di framework che supportano il SSR sono: Angular Universal, Next.js per React e Nuxt per Vue, che verrà illustrato nel capitolo 2.

[11] Google developers - Core Web vitals - Al 10 maggio 2023, la durata massima ammissibile per il FCP è di 2.5s.

1.8 Servizi cloud e containerizzazione

Le tecniche di rilascio di applicazioni Web si sono evolute di pari passo alle tecnologie di sviluppo.

Il primo modello era quello *monolitico*, in cui l'applicazione Web viene distribuita su un server fisico che necessita di configurazioni manuali.

Poi, attorno all'inizio degli anni 2000, sono emersi i primi **provider di cloud**, come Amazon Web Services, Google Cloud Platform, Microsoft Azure e IBM Cloud, che non solo offrivano servizi di hosting di *virtual private server*, traducendo quindi il paradigma monolitico su macchine virtuali, ma anche servizi di *infrastrucure as a service*, permettendo ai progettisti di applicazioni Web di modificare le risorse di calcolo e di archiviazione secondo necessità e di automatizzare il processo di rilascio.

Una tecnologia in particolare si è affermata come standard per la distribuzione di applicazioni Web attraverso l'infrastruttura cloud, il **container**, con la sua implementazione più popolare, Docker.

Docker è un sistema di virtualizzazione di risorse a livello di sistema operativo, che permette di isolare i processi di un'applicazione in un ambiente chiuso che condivide il kernel del sistema operativo host. Questa tecnica è diversa da quelle di virtualizzazione classiche, che mirano ad emulare un intero sistema operativo, come fa ad esempio KVM, o un intero sistema hardware, come fa QEMU. Docker è più leggero e più veloce, e permette di avere un ambiente di esecuzione riproducibile e portabile, che può essere distribuito su qualsiasi macchina che abbia Docker installato. [12]

Il ruolo degli informatici coinvolti nello sviluppo di applicazioni Web si è quindi differenziato tra gli *architetti di cloud* e gli *ingegneri di sviluppo*. Tuttavia, una volta che l'infrastruttura è pronta, il processo di rilascio di un'aggiornamento dell'applicazione si semplifica notevolmente grazie ai metodi di **continuous integration** come Travis CI, Circle CI e Github Actions, che permettono di automatizzare il processo di build e di test, e di rilasciare su cloud con la stessa facilità con cui si fa un commit sul repository di codice.

[12] Docker - What is a container? In questo articolo vengono comparate le tecnologie di virtualizzazione tradizionali con Docker.

Capitolo 2

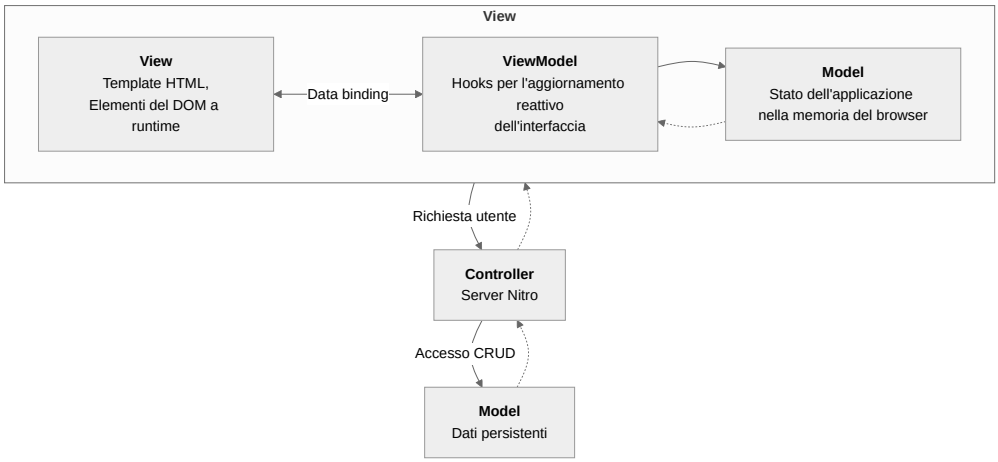
Descrizione delle tecnologie

In questo capitolo si illustrano due particolari tecnologie per la realizzazione di applicazioni Web: Nuxt e TypeORM. Sono state scelte tra le molte alternative disponibili per il loro uso diffuso e consolidato nel settore, e perché esemplificano una naturale continuazione delle linee evolutive descritte nel capitolo precedente fornendo una soluzione alle problematiche affrontate, e per altre ragioni che saranno discusse in seguito.

2.1 Nuxt

Nuxt è un framework per applicazioni Web, avviato come progetto Open source da Alexandre e Sebastien Chopin e Pooya Parsa nel 2016, che continua ad essere mantenuto attivamente su Github da un team di sviluppatori che accettano contributi, all'indirizzo github.com/nuxt/nuxt.

Nuxt si propone di risolvere i problemi di performance, di ottimizzazione e di accessibilità delle applicazioni basate su componenti con il suo migliorato sistema di frontend, ma anche di fornire un ambiente di sviluppo flessibile, per facilitare la scalabilità e la manutenibilità del codice backend. Si possono infatti realizzare applicazioni **fullstack** secondo il pattern MVC, in cui la view è implementata con Vue.js ed il controller con *Nitro*, un server http fatto su misura per Nuxt.



L'architettura generale di una applicazione Nuxt. Si noti che la View adotta a sua volta il pattern *MVVM* quindi si hanno due modelli dei dati con interfacce potenzialmente distinte. Infatti nel modo tradizionale di usare Vue, backend e frontend potrebbero essere viste come due applicazioni a bassa coesione (basti pensare a come potrebbero essere realizzate in due linguaggi di programmazione differenti) ed alto accoppiamento (nel senso che un cambiamento da un lato potrebbe richiedere un cambiamento dall'altro lato del sistema, per mantenere la coerenza). Nuxt si occupa appunto di gestire la **comunicazione tra i due models**: il model dei dati persistenti ed il model dell'applicazione che esegue nel browser, in modo da ottenere *loose coupling* e *high cohesion*.

Lo slogan di Nuxt è "The Intuitive Vue Framework", che è in accordo con il suo obiettivo di semplificare la creazione di applicazioni Web fornendo un'infrastruttura preconfigurata e pronta all'uso. In questo modo lo sviluppatore può concentrarsi da subito sulla logica dell'applicazione, piuttosto che sulla configurazione del progetto. È quindi ricalcato il punto di vista di David Heinemeier Hansson su Rails, il framework per applicazioni Web per Ruby che ideò nel luglio 2004, per il quale sosteneva il principio "convention over configuration"[1].

Nonostante questo, Nuxt utilizza internamente tecnologie raffinate, come Typescript e Vite, che consentono di scrivere codice robusto. Con Nuxt si possono realizzare applicazioni di vario genere, come siti vetrina, blog, documentazioni o wiki, e-commerce, dashboard gestionali, piattaforme di social networking, applicazioni mobile-first, etc…

La repository di sviluppo di Nuxt è organizzata secondo il modello di *monorepo*, quindi include pacchetti funzionanti in maniera disaccoppiata, ma che sono usati tutti in maniera coesa all'interno del sistema Nuxt. La versione corrente è la **3.15**, rilasciata nel dicembre 2024. La struttura del monorepo è la seguente:

- `packages/nuxt` è il core del framework.
- `packages/nuxi` è lo strumento da linea di comando per la creazione di nuovi progetti, ora spostato su `github.com/nuxt/cli`.
- `packages/schema` contiene le definizioni dei tipi di dati utilizzati.
- `packages/kit` è un toolkit per la creazione di moduli aggiuntivi.
- `packages/test-utils` contiene degli script per il testing di unità.
- `packages/vite` è una fork di Vite, un bundler per gli script di frontend, usato di default da Nuxt.
- `packages/webpack` è una fork di Webpack, un'altro bundler per gli script di frontend che si può scegliere in alternativa a Vite.
- `docs` è la documentazione ufficiale, scritta sotto forma di sito Web statico, usando Nuxt stesso.

Oltre a modificare la monorepo, gli sviluppatori Open source sono invitati a creare moduli per estendere le Nuxt con funzionalità non essenziali, ma idonee per l'interoperabilità con altri software. Questi moduli possono essere pubblicati su Npm come pacchetti, con `@nuxt/kit` come dipendenza, ed al Marzo 2025 se ne contano più di 200[2].

[1] Wikipedia - Convention over configuration
[2] Moduli supportati ufficialmente da Nuxt.

La versione vanilla di Nuxt propone un'intelaiatura che include una command line interface con cui si definisce il funzionamento del backend, che determina il modo in cui il frontend verrà mostrato agli utenti. Il frontend, a sua volta, è in comunicazione con il backend per ottenere dati aggiornati.

In questo schema sono mostrate queste parti e le loro interazioni:

2.1.1 Command line interface

L'ecosistema Nuxt fa uso di un programma invocabile da linea di comando chiamato *nuxi*. È installabile globalmente su un sistema provvisto di Node eseguendo `npm i -g @nuxt/cli`, e dispone di vari sotto-comandi per la gestione del progetto. È consigliato usare `npx nuxi <sotto-comando>` per evitare conflitti tra le versioni dei pacchetti installati localmente e globalmente: anteponendo "npx" si userà, se presente, la versione locale `node_modules/@nuxt/cli`.

`nuxi init <nome-progetto>`

È il comando per avviare un nuovo progetto nella directory `./<nome-progetto>`. Eseguendolo si dovrà scegliere il sistema di gestione dei pacchetti, che riguarderà il modo con il quale Nuxt ed anche gli agli altri pacchetti di terze parti saranno installati, e può essere tra:

- **Npm**: Il classico package manager di Node, solitamente installato assieme ad esso scegliendo il pacchetto `node` nelle repository delle maggiori distribuzioni Linux, e disponibile di default nelle immagini Docker ufficiali di Node.
- **Pnpm**: Un package manager alternativo a npm, progettato per migliorare le performance e ottimizzare l'utilizzo dello spazio su disco rispetto a npm, preferito per lo sviluppo locale.
- **Yarn**: Un altro package manager alternativo a npm, sviluppato in Facebook nel 2016.
- **Bun**: Con questa opzione si sceglie di usare una runtime diversa da Node: Bun, più efficiente in alcune operazioni di I/O, compatibile con le API Node e i suoi pacchetti di terze parti.
- **Deno**: Un'altra runtime JavaScript che offre supporto nativo a Typescript, ma non è del tutto compatibile con alcuni pacchetti npm.

Subito dopo c'è la scelta **Initialize git repository**, che eseguirà `git init` se selezionata. Nella trattazione che segue adotteremo Npm come package manager e Git per il controllo di versione.

La directory `./nome-progetto` sarà indicata come `~`[3], e conterrà i seguenti:

```
.git/               # Versioni dei file del progetto
.nuxt/              # Files temporanei usati dal server di sviluppo
.output/            # Files generati durante la build per la produzione
node_modules/       # Librerie di Nuxt e di terze parti
public/             # Risorse statiche da distribuire con l'applicazione
```

[3]Nel contesto di sistemi Unix-like, la tilde `~` è un alias per la directory home dell'utente corrente. Nei files di un'app Nuxt invece indica la directory radice del progetto.

```
    robots.txt      # File di configurazione per i motori di ricerca
    favicon.ico     # Icona del sito
server/             # Directory preposta al codice riservato al server
    tsconfig.json   # Configurazione del compilatore Typescript per il backend
.gitignore          # Lista dei file da ignorare durante il versionamento
app.vue             # Entry point dell'applicazione
nuxt.config.ts      # File di configurazione di Nuxt
package-lock.json   # Albero delle versioni delle dipendenze
package.json        # Lista delle dipendenze e dei comandi di build
README.md           # Documentazione del progetto
tsconfig.json       # Configurazione del compilatore Typescript per il frontend
```

`nuxi add`

Una volta inizializzato il progetto, questo è il comando per aggiungere funzionalità all'app. Prende come terzo argomento il tipo di template da aggiungere, che può essere tra:

- **app**: Il componente Vue che fa da entry point dell'applicazione. È già presente di default in ogni progetto Nuxt, ma può essere sovrascritto con questo comando.
- **page**: Una pagina Web, che sarà accessibile alla rotta `/<nome-pagina>`.
- **layout**: Un layout Vue, cioè un componente che definisce la struttura di una o più pagine. È un modo di riutilizzare il codice HTML e CSS in più parti dell'applicazione.
- **component**: Un componente Vue, riutilizzabile in tutte le pagine o layout.
- **error**: Un componente Vue che sarà mostrato in caso di errore.
- **middleware**: Un middleware, cioè una funzione che può essere eseguita prima di caricare una pagina, lato server o lato client.
- **composable**: Una funzione che può essere usata in uno o più componenti Vue. È un modo per riutilizzare la logica di business *stateful* in più parti dell'applicazione.
- **plugin**: Uno script Typescript che viene eseguito prima di inizializzare l'applicazione Vue. Utile per l'inizializzazione di componenti software di terze parti. A differenza dei middleware, i plugin vengono eseguiti solo una volta, all'avvio dell'applicazione.
- **api**: Un endpoint API, che sarà accessibile alla rotta `/api/<nome-endpoint>`. Utile per la comunicazione tra frontend e backend.
- **server-route**: Un endpoint API, che sarà accessibile alla rotta `/<nome-endpoint>`.
- **server-middleware**: Un middleware, simile a quelli di Express, che si interpone richiesta e risposta. Utile per l'autenticazione, la gestione delle sessioni, la compressione dei dati…
- **server-plugin**: Uno script Typescript che viene eseguito prima di inizializzare il server Nitro. Utile per l'inizializzazione di componenti software di terze parti.
- **server-util**: Un modulo Typescript importato automaticamente in ogni file di tipo server.
- **module**: Con questa opzione si crea un modulo Nuxt per sperimentarlo, e che potrà essere utilizzato anche in altri progetti.

Ogni aggiunta corrisponde ad un nuovo file che verrà creato nella directory corrispondente, provvisto di un *boilerplate*[4], che sarà possibile modificare per adattarlo alle proprie esigenze.

[4] Cioè del codice ripetuto frequentemente.

`nuxi dev`

Una volta aggiunte le prime funzionalità si può lanciare il server di sviluppo, che permette di testare l'applicazione in locale. Di default il server è accessibile alla rotta `http://localhost:3000`, ma si può cambiare la porta con l'opzione `--port <numero-porta>`. Il server di sviluppo è dotato nativamente di *hot reloading*, cioè la capacità di ricaricare automaticamente la pagina senza smaltire lo stato di esecuzione Javascript quando si salvano i file del progetto, in modo da velocizzare il feedback del sistema al programmatore.

`nuxi devtools`

Abilita o disabilita l'iniezione degli script Devtools nell'app Vue, quando è lanciata con `nuxi dev`. Sono un set di strumenti il debugging di applicazioni Nuxt, aggiuntivi a quelli già presenti nei browser moderni[5].

Viene aggiunto un elemento html alla pagina, nel quale sono presenti diverse sezioni che mostrano informazioni di profilazione dell'app in sviluppo, tra cui:

- **Pages**: Lista delle pagine dell'applicazione, con la possibilità di navigare tra di esse.
- **Components**: Lista dei componenti Vue inseriti nel bundle, con riferimenti e dipendenze.
- **Components tree**: Albero dei componenti Vue della pagina corrente. Fornisce una visualizzazione più ordinata rispetto ai devtools del browser, dove si possono vedere solamente gli elementi risultanti dal rendering Vue.
- **Imports**: Lista dei composables inseriti nel bundle.
- **Modules**: Lista dei moduli utilizzati dall'applicazione, lato server o client.
- **Assets**: Risorse statiche usate dall'applicazione, come immagini, font, icone, etc…
- **Open Graph**: Metadati Open Graph[6] della pagina corrente, utili per la condivisione sui social network.
- **Timeline**: Un grafico che mostra il tempo di rendering delle pagine, dei componenti e dei moduli.
- **Hooks**: Lista dei hooks, cioè delle funzioni che vengono eseguite in determinati momenti del ciclo di vita dell'applicazione e dei singoli componenti.
- **Server routes**: Un modo per visualizzare le rotte del server Nitro e fare delle richieste di test, con possibilità di aggiungere parametri GET, body POST, header, cookies e di simulare la provenienza della richiesta dall'app.
- **Inspect**: In questa sezione il codice dei file Vue e Typescript per il frontend viene riportato con tutti gli stage di compilazione, fino al codice Javascript finale eseguito dal browser.

`nuxi module`

Ha due ulteriori sottocomandi, `search` e `add`, che permettono di cercare e aggiungere moduli Nuxt, tra quelli ufficialmente supportati[7].

[5] Come quelli di Firefox, dei derivati di Chromium, di Safari ed di Edge.

[6] Open Graph Protocol - Protocollo per l'inserimento di metadati nelle pagine Web, che saranno mostrati come copertina quando la pagina viene condivisa sui social network.

[7] Moduli supportati ufficialmente da Nuxt.

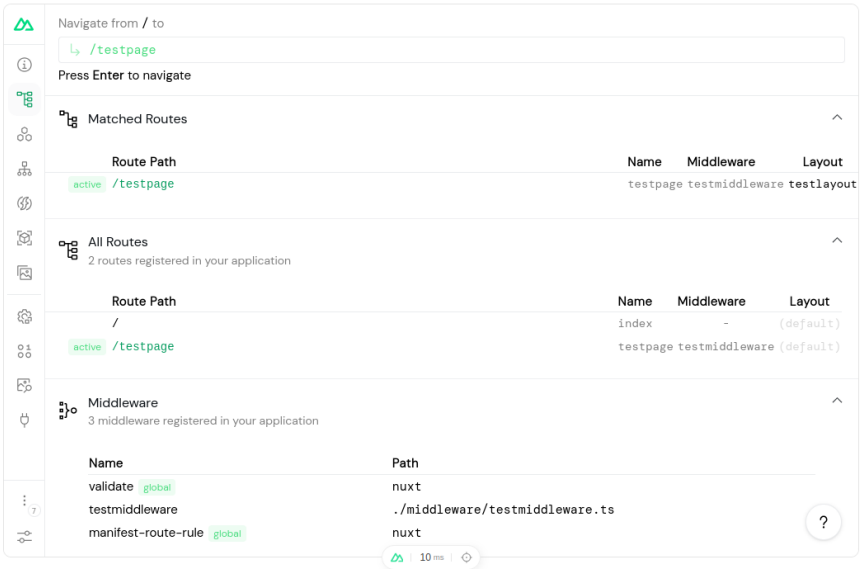


Figura 2.1: Devtools di Nuxt. In questa sezione è mostrato la lista delle pagine agganciate al Vue-router, i middleware e i layout per ogni pagina. È presente inoltre un indicatore che mostra come il rendering della pagina `testpage` ha impiegato 10ms.

`nuxi typecheck`

Consente di eseguire il controllo statico del codice Typescript, per trovare errori di sintassi e di logica prima di eseguire la build dell'applicazione, anche nei file Vue. Richiede l'installazione di `vue-tsc` come dipendenza di sviluppo.

`nuxi test`

Esegue i test definiti in `~/tests`. Richiede l'installazione di `@nuxt/test-utils` come dipendenza di sviluppo. In questo modo si possono avviare i testi di

- **Unità:** sono i test che verificano il comportamento di una singola funzione o di un singolo componente secondo la previsione del programmatore. Sono implementati con `vitest`, di default, o `jest`.
- **Componenti:** permettono di verificare il corretto funzionamento di un singolo componente e dei composable ad esso associati.
- **Integrazione:** si tratta di test che verificano il corretto funzionamento di più componenti insieme. Questo tipo di test garantisce che, ad un'aggiunta di un nuovo componente, non si verifichino errori di rendering o di logica con i componenti già esistenti.
- **End-to-end (E2E):** questo tipo di test simula l'interazione di un utente con l'applicazione, attraverso un browser virtuale, implementato con `playwright` o `puppeteer`. Questo tipo di test garantisce che l'applicazione sia accessibile e usabile da un utente reale, mitigando i problemi di accessibilità.

```
describe("CounterWithComposable Component", () => {
    it("è renderizzato correttamente", () => {
        const wrapper = mount(CounterWithComposable);
        expect(wrapper.text()).toContain("Count: 0");
    });

    it("incrementa il valore quando il bottone Increment è premuto", async () => {
        const wrapper = mount(CounterWithComposable);
        await wrapper.find("button:first-of-type").trigger("click");
        expect(wrapper.text()).toContain("Count: 1");
    });
});
```

Esempio di test di un componente Vue con `vitest`. Il test non fa asserzioni sulla struttura dati interna del componente, ma verifica che esso sia renderizzato e funzioni correttamente.

`nuxi build`

Compila il codice Typescript e genera i file necessari per la distribuzione dell'applicazione. I file generati sono salvati nella directory `~/.output`, e possono essere distribuiti su un server Node, Deno o Bun per la produzione. L'albero delle dipendenze viene ridotto al minimo con una procedura chiamata *tree-shaking* e le dipendenze necessarie per l'esecuzione dell'applicazione vengono copiate in `~/.output/server/node_modules`.

`nuxi generate`

Fa una build dell'applicazione pre-renderizzando tutte le route raggiungibili, anche quelle forzate dai valori cablati nei componenti Vue: il compilatore percorre non solo le pagine puramente statiche (senza richieste ad api) ma anche quelle che fanno richieste ad API senza dei parametri forniti a tempo di esecuzione dall'utente. Verranno generati dei files HTML statici per ogni rotta di questo tipo, per essere serviti da un server frontend statico, in `~/.output/public`, e di questa directory verrà creato un link simbolico in `~/dist`.

`nuxi cleanup`

Rimuove i file temporanei e i file generati durante la build.

2.1.2 Frontend

Nuxt adotta delle convenzioni per il frontend: i file che definiscono le pagine accessibili all'utente sono organizzate in una struttura gerarchica di directory. Ogni pagina può essere inclusa in un layout, che definisce la struttura generale della pagina, e può usare dei middleware, che sono delle funzioni che vengono eseguite prima di caricare la pagina. Ogni componente di cui le pagine sono composte può essere definito in un file separato, per favorire il riutilizzo del codice. Tutti i file Vue che seguono le convenzioni di Nuxt sono importati automaticamente.

Come per un'applicazione Vue tradizionale, è presente un entry point, `app.vue`:

```vue
<template>
    <NuxtLayout>
        <NuxtPage />
    </NuxtLayout>
</template>
```

Il template dell'intera app racchiude, tra `NuxtLayout`, il componente `NuxtPage`, che rappresenta la pagina corrente.

Pages

Per definire una pagina è sufficiente creare un file Vue nella directory `~/pages`. La relativa pagina sarà accessibile alla rotta secondo le regole di *file system routing*:

- `index.vue` è la pagina principale, accessibile alla rotta `/`.
- Ogni file rappresenta un endpoint di rotta.
- Le directory influenzano la rotta della pagina, tranne quelle tra parentesi tonde `()`.
- Si possono aggiungere parametri alla rotta con `[nome]` o `[...nome]`, e questi sono disponibili come variabili nel componente Vue.

```
pages/
    index.vue           # Pagina principale, accessibile alla rotta /
    about.vue           # Pagina accessibile alla rotta /about

    gruppo/             # Gruppo di pagine, influenza la rotta
        pagina-1.vue    # Pagina accessibile alla rotta /gruppo/pagina-1
        pagina-2.vue    # Pagina accessibile alla rotta /gruppo/pagina-2

    (gruppo)/           # Gruppo di pagine, non influenza la rotta
        pagina-1.vue    # Pagina accessibile alla rotta /pagina-1
        pagina-2.vue    # Pagina accessibile alla rotta /pagina-2

    [id].vue            # Pagina accessibile a /<id>.
    [...id].vue         # Pagina accessibile a /<id[0]>/<id[1]>/<id[2]>...
    gruppo-[nome]/      # Gruppo di pagine con variabile <nome>, influenza la rotta
        pagina-1.vue    # Pagina accessibile a /gruppo-<nome>/pagina-1
```

I file Vue che definiscono le pagine sono composti da tre parti: il template, lo script e lo stile.

```vue
<script setup lang="ts">
    definePageMeta({
        middleware: ["auth"],
        layout: "home",
    });
</script>

<template>
    <h1>Pagina iniziale</h1>
```

```
10      <NuxtLink to="/about">Vai alla pagina about</NuxtLink>
11  </template>
12
13  <style scoped>
14      /* Stili CSS */
15  </style>
```

Template Nel template si definisce la struttura della pagina, con i tag HTML e i componenti Vue. Si possono usare le direttive di Vue per iterare sui dati, condizionare il rendering di elementi, gestire gli eventi e le classi CSS.

Per utilizzare appieno il sistema di routing di Nuxt, è sconsigliato usare i classici anchor tags `<a>` per navigare tra diverse pagine. Questo perché non sempre è necessario fare una nuova richiesta HTTP al server del frontend per richiedere la prossima pagina, che potrebbe essere già presente nella memoria del browser sotto forma di componenti Vue.

Si può usare invece il componente `<NuxtLink>`, che accetta l'attributo `to` con il nome della rotta da raggiungere, e che si occuperà di cambiare pagina nella maniera più efficiente:

- tramite il **router Vue**, con `history.pushState` e aggiornando il Virtual DOM con rimozione dei componenti non necessari ed aggiunta di quelli nuovi. All'aggiornamento del Virtual DOM segue l'aggiornamento del DOM reale, in base alle differenze tra i due.
- tramite il **prefetching** delle rotte percorribili dalla pagina corrente, cioè il caricamento in background delle risorse necessarie delle possibili pagine successive, prima che l'utente clicchi sul relativo link.

I protocolli permessi sono:

- la **client-side navigation**, cioè la navigazione tra le pagine senza fare richieste HTTP al server se i dati necessari sono già disponibili nel browser.
- il **data fetching**, cioè la navigazione tra le pagine con richieste HTTP efficienti al server, che risponde non con l'intera nuova pagina HTML, ma con solo i dati necessari per renderizzarla a partire dallo stato della pagina già disponibile nel browser. Nuxt suddivide le pagina in *chunks* Javascript, che vengono scaricati solo quando è necessario. I nuovi componenti Vue porteranno con sé i dati aggiornati dalle API (per una spiegazione di questo confrontare il paragrafo Modalità di rendering del frontend).

È una buona pratica quella di guarnire i template delle pages con contenuti HTML e riferimenti ad altri componenti Vue, delegando a questi la logica di presentazione, nonostante anche nelle pages si possano inserire delle direttive Vue.

Script Nello script si definisce la logica della pagina, come la gestione degli eventi, la comunicazione con il backend, la gestione dello stato dell'applicazione e la definizione dei metadati della pagina. Le pratiche consigliate sono di mantenere gli script delle pagine leggeri il più possibile, delegando la logica di business ai componenti Vue e ai composables. Nell'esempio è inserito uno script con `setup` per abilitare la *composition api* di Vue e `lang="ts"` per usare Typescript. Inoltre, è fatto uso dell'hook `definePageMeta`, che permette di definire:

- `name? string`: il nome della pagina, di default generato dalle regole di routing.
- `path? string`: si usano espressioni regolari per definire rotte dinamiche.
- `props? RouteRecordRaw["props"]`: definisce le proprietà della pagina, che possono essere passate come parametri GET o POST.
- `alias? string | string[]`: definisce un alias per la pagina, cioè un altro nome con cui è accessibile.
- `layout? LayoutKey`: definisce il layout della pagina, cioè il componente Vue che definisce la struttura della pagina. Il tipo `LayoutKey` tipizza come string literal i nomi dei layout disponibili.
- `middleware? MiddlewareKey | Array<MiddlewareKey>`: definisce i middleware da eseguire prima di caricare la pagina. Il tipo MiddlewareKey tipizza come string literal i nomi dei middleware disponibili.
- `pageTransition? boolean | TransitionProps`: definisce se la transizione tra le pagine deve essere animata, secondo proprietà che usano regole CSS.

Style Tra i tag `<style>` si definiscono i fogli di stile della pagina, con regole CSS che saranno applicate ai tag HTML e ai componenti Vue. Si può usare la direttiva `scoped` per limitare l'ambito delle regole CSS al solo componente Vue corrente, in modo da evitare conflitti con altri stili definiti in altri componenti. Inoltre Nuxt supporta i preprocessori CSS come Sass, Less e Stylus, che permettono di scrivere regole CSS più complesse e riutilizzabili.

```
<style lang="scss">
    @use "~/assets/scss/main.scss";
</style>
```

Components

Nuxt supporta componenti Vue globali, cioè che possono essere usati in più pagine o layout. Sono definiti in file separati, nella directory `~/components`, e sono importati automaticamente nei file Vue che li usano, solamente se sono usati. Seguono le regole di sintassi Vue, di cui si ricordano le direttive:

- `{{ string }}`: la così detta *moustache syntax*, che permette di interpolare il valore di una variabile all'interno di un template.
- `v-bind:attribute="value"`: permette di associare un attributo HTML a un valore, in modo che il valore venga interpolato nell'attributo.
- `v-if="boolean"`: permette di condizionare il rendering di un elemento in base al valore di una variabile booleana.
- `v-for="item in array"`: permette di iterare su un array di elementi e di renderizzare elementi HTML per ogni elemento dell'array.
- `v-on:<event>="handler"`: permette di associare un evento a un handler, cioè una funzione che verrà eseguita quando l'evento viene emesso. C'è una forma abbreviata per gli eventi più comuni, come `@click`, `@input`, `@submit`, etc…
- `v-model="variable"`: permette di creare un *two-way binding* tra una variabile e un input: un componente padre può passare una variabile a un componente figlio con `<Child v-model="value" />`,

e il componente figlio può accedere al valore usando `const value = defineModel()` nello script. Il binding è bidirezionale, quindi se il valore cambia nel componente figlio, allora cambierà anche nel componente padre.

e i metodi di reattività:

- `ref(value)`: definisce una variabile reattiva, che può essere modificata e che notificherà ai componenti che la usano di aggiornarsi tramite il sistema di reattività di Vue. È da preferirsi quando si trattano valori primitivi (come stringhe o numeri), sebbene supporti anche oggetti e array, che osserva con un solo livello di profondità. La variabile reattiva è accessibile tramite la proprietà `.value`.
- `reactive(object)`: definisce un oggetto reattivo (ma non un valore primitivo), ma il metodo di accesso è diverso rispetto ad un ref. È da preferirsi quando si intende modificare un oggetto particolarmente complesso, perché è ottimizzato per la modifica di più proprietà contemporaneamente.
- `watch(variable, (newValue, oldValue?) => void)`: permette di eseguire una funzione quando una variabile reattiva viene modificata
- `computed(() => value)`: permette di definire una variabile calcolata, che viene aggiornata automaticamente quando le variabili reattive di cui dipende vengono modificate. Le variabili reattive non vengono passate come argomenti, ma vengono osservate automaticamente dal sistema di reattività di Vue.

In Nuxt sono disponibili dei componenti *built-in*:

- `<ClientOnly>`: permette di renderizzare un componente solo lato client, cioè solo quando l'applicazione è eseguita sul browser.
- `<DevOnly>`: permette di renderizzare un componente solo in ambiente di sviluppo.
- `<NuxtImg>`: renderizza con efficienza immagini con supporto per il lazy loading e per la generazione di immagini responsive.
- `<NuxtPicture>`: ha lo stesso funzionamento di `<NuxtImg>`, ma usa internamente il tag `<picture>` per supportare immagini con formati diversi.
- `<NuxtLoadingIndicator>`: renderizza un indicatore di caricamento che può essere personalizzato.

Layouts

I layout Nuxt sono componenti che definiscono la struttura comune a una o più pagine. La pratica consigliata è infatti quella di non inserire script, ma solo template e stili CSS: se c'è necessità di condividere logica tra più pagine, è preferibile usare composables o middleware. I layout sono definiti in file separati nella directory `~/layouts`, e sono importati automaticamente nei file Vue che li usano.

Un esempio di layout, dove la pagina è montata in `<slot />`, tra un header e un footer:

```
1 <template>
2     <div>
```

```
        <AppHeader />
        <slot />
        <AppFooter />
    </div>
</template>
```

Composables

Un composable Vue è una funzione riutilizzata in più componenti che incapsula della logica *stateful*. Un esempio è il composable `useCounter` che gestisce lo stato di un contatore e la sua operazione di incremento:

```
export const useCounter = () => {
    const count = ref(0);
    const increment = () => {
        count.value++;
    };
    return { count, increment };
};
```

Nuxt offre un sistema di importazione automatica dei composables definiti in `~/composables`, oltre che di quelli built-in:

- **useRoute**: permette di accedere alla rotta corrente, ai parametri GET e POST, ai parametri dinamici e ai parametri query. Con `useRoute().params.<nome parametro>` si accede ai parametri dinamici delle pagine
- **useRouter**: permette di accedere al router Vue, per navigare tra le pagine dell'applicazione. Contiene le informazioni di cronologia del sito e permette di navigare tra le pagine con `useRouter().push(<nome rotta>)`.
- **useHead**: permette di modificare i metadati della pagina, come il titolo, la descrizione, le parole chiave, l'immagine di copertina e il tipo di contenuto, per migliorare l'indicizzazione sui motori di ricerca.
- **useSeoMeta**: consente di modificare metadati aggiuntivi, come quelli di open graph[8].
- **useError**: restituisce un errore, se presente, e permette di gestirlo in maniera personalizzata.
- **useState**: permette di definire una variabile reattiva condivisa tra server e client: non ci saranno problemi di sincronizzazione o di duplicazione degli eventi al caricamento della pagina nel browser. È da preferirsi all'utilizzo di `ref` o `reactive` quando si tratta di variabili che devono essere condivise tra server e client.
- **useAsyncData**: permette di recuperare dati in modo asincrono e reattivo all'interno di un componente o di una pagina. Accetta una funzione che restituisce una Promise e gestisce automaticamente lo stato di caricamento, errore e dati ricevuti. Utile per chiamate API e operazioni asincrone.

[8] Open Graph Protocol - Protocollo per l'inserimento di metadati nelle pagine Web, che saranno mostrati come copertina quando la pagina viene condivisa sui social network.

- **useLoadingIndicator**: fornisce un indicatore di caricamento globale per l'applicazione. Può essere utilizzato per mostrare una barra di progresso o un'animazione durante il caricamento di pagine e dati.
- **useCookie**: permette di leggere, scrivere e rimuovere cookie in modo reattivo, sia lato client che lato server. Utile per la gestione dello stato dell'utente, autenticazione e preferenze.

Middleware

I middleware del frontend di Nuxt sono file Typescript che espongono una funzione `defineNuxtRouteMiddleware` che accetta due parametri, `to` e `from`, che rappresentano la rotta corrente e la rotta precedente. Questa funzione può essere usata per eseguire operazioni prima di caricare una pagina, come la verifica dell'autenticazione dell'utente, la gestione degli errori, il prefetching delle risorse o il logging.

```
export default defineNuxtRouteMiddleware((to, from) => {
    const user = useCookie("user"); // Legge il cookie "user" per verificare
     ↪  l'autenticazione

    if (!user.value) {
        return navigateTo("/login"); // Se l'utente non è autenticato, reindirizza alla
         ↪  pagina di login
    }
});
```

2.1.3 Backend e API fetching

Nitro è il motore server-side di Nuxt, progettato per essere flessibile, performante ed indipendente dal framework frontend. Il suo progetto github è portato avanti da un team di sviluppatori che si occupano di mantenere il codice e di risolvere i bug all'indirizzo github.com/nitrojs/nitro, e la sua versione più recente è la **2.11**.

Nitro è in grado di esporre API RESTful e di gestire richieste HTTP e WebSocket. Può essere eseguito su runtime Node, Bun e Deno e dispone di un motore di rendering Vue integrato, che permette di preparare l'HTML da inviare al client in maniera efficiente.

Come per il file system routing delle pagine, anche il backend di Nuxt segue delle convenzioni per la definizione delle rotte API. I file che definiscono gli endpoint API sono organizzati in una struttura gerarchica di directory, e sono importati automaticamente nei file che li usano. Nella directory `~/server` ogni file in `api/` o in `routes/` rappresenta un endpoint API, sulla quale può essere specificato il metodo HTTP, i parametri GET e POST, e il corpo della risposta.

```
server/
    api/
        users.ts                    # GET /api/users
        users/
            byLastName.post.ts      # POST /api/users/byLastName
            [id].ts                 # GET /api/users/<id>
        [...].ts                    # GET /api/*
    routes/
        messages.get.ts             # GET /messages
```

```
        messages.post.ts            # POST /messages
    middleware/
        log.ts
```

Endpoint API

Per definire il comportamento di un endpoint API, si esporta una funzione `defineEventHandler(async? (event)` che accetta una funzione asincrona di un oggetto `event` e restituisce una risposta. L'oggetto `event` contiene le informazioni della richiesta HTTP, come il metodo, i parametri GET e POST, i cookie, l'indirizzo IP del client, e il corpo della richiesta.

```ts
export interface UsersByLastName {
    lastName: string;
}

export default defineEventHandler(async event => {
    const body = await readBody<UsersByLastName>(event);
    const lastName = body.lastName;

    const users = await // ottenimento di utenti con il cognome specificato dal database.

    return {
        status: 200,
        body: { users }
    }
})
```

Per richieste GET si possono ottenere i parametri con `getQuery(event: H3Event)` e per richieste POST si può ottenere il body con `readBody<T>(event: H3Event)`, che restituisce un oggetto di tipo `T` deserializzato dal body della richiesta. Il generic `T` è opzionale, ma è buona pratica specificare il tipo dell'oggetto che ci si aspetta di ricevere, per evitare errori di tipo. Rimane però al programmatore la responsabilità di usare la stessa interfaccia tra frontend e backend.

A partire dagli issue 23009 di Nuxt e alla discussione 235 di Nitro sono state avanzata delle proposte per rendere completamente type-safe gli scambi di oggetti JSON tra l'app in esecuzione nel browser e il server Nitro. È in corso il lavoro di implementazione del modulo `nuxt-server-fn`[9] che permette di condividere delle funzioni Typescript tra client e server. Questo modulo rileva le funzioni esportate nei files Typescript sotto `~/server/functions/`, come la seguente:

```ts
export async function getUsersByLastName(lastName: string): User[] {
    const users = await // ottenimento di utenti con il cognome specificato dal database.
    return users;
}
```

per renderle disponibili con l'utilizzo del composable sperimentale `useServerFunctions()` nel frontend:

[9] Disponibile nella apposita repository github.

```vue
<script setup lang="ts">
    const { getUsersByLastName } = useServerFunctions();
    const users = reactive<User[]>([]);
    const lastName = ref("");

    watch(input, async (newValue) => {
        users.value = await getUsersByLastName(lastName.value);
    });
</script>
```

Questo snippet aggiorna la variabile reattiva `users` al cambiamento di `lastName`, facendo immediatamente una richiesta al server Nitro per ottenere gli utenti con il cognome specificato. Il tipo del parametro `lastName` è imposto dal parametro della funzione.

Fetching

Per fare richieste HTTP dal frontend di Nuxt si può usare il composable `useFetch`, che permette di fare richieste HTTP e di gestire lo stato di caricamento, errore e dati ricevuti. È un wrapper attorno ad `useAsyncData`, ed internamente utilizza `$fetch`, una utility di Nuxt basata su ofetch[10] che permette di fare richieste HTTP, sia dal client che dal server ed include parser JSON automatico che è in grado di serializzare e de-serializzare oggetti in maniera efficiente.

`useFetch` è un composable asincrono, wrapper attorno a `useAsyncData`, che permette di fare richieste HTTP in maniera reattiva e di gestire lo stato di caricamento, errore e dati ricevuti. Accetta un URL e un oggetto di opzioni, e restituisce un oggetto con le proprietà `data`, `pending` e `error`. Queste proprietà sono reattive, e cambiano automaticamente quando cambia lo stato della richiesta, così è possibile aggiornare l'interfaccia appena i dati sono disponibili.

```vue
<script setup lang="ts">
    const usersByLastName = reactive<UsersByLastName>({ lastName: "" });

    const { data, pending, error } = await useFetch("/api/users/byLastName", {
        method: "POST",
        body: usersByLastName,
        watch: [usersByLastName],
        lazy: true,
    });
</script>

<template>
    <input v-model="usersByLastName.lastName" />
    <div v-if="pending">Caricamento...</div>
    <div v-if="error">Errore: {{ error }}</div>
    <div v-if="data">
        <ul>
            <li v-for="user in data.users">
                {{ user.firstName }} {{user.lastName}}
```

[10] Un progetto portato avanti dalla cooperativa Unjs alla su github.com/unjs/ofetch.

```
            </li>
        </ul>
    </div>
</template>
```

L'esempio sopra mostra una riscrittura della server function `getUsersByLastName` con `useFetch`. Al cambiamento del valore di `usersByLastName.lastName`, `useFetch` fa una richiesta POST all'endpoint `/api/users/byLastName` con il corpo della richiesta `usersByLastName`, e aggiorna le variabili reattive `data`, `pending` e `error` con i dati ricevuti, lo stato di caricamento e gli errori. Il template mostra un messaggio di caricamento se `pending` è `true`, un messaggio di errore se `error` è definito o la lista degli utenti se `data` è definito.

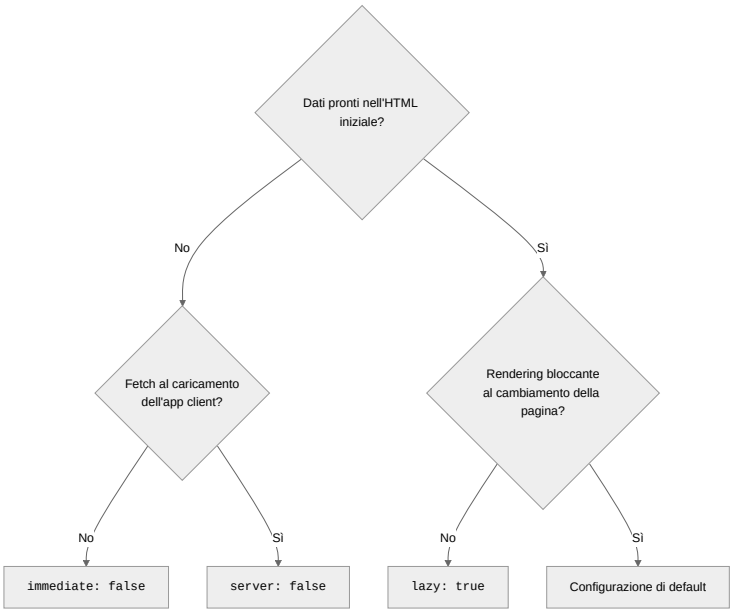
Oltre al primo argomento che è una stringa con la rotta dell'API (ha tipizzazione forte e ammette solo rotte valide), le opzioni di `useFetch` sono:

- `headers?: Record<string, string>`: permette di inserire header personalizzati nella richiesta.
- `timeout?: number`: il tempo massimo di attesa per la risposta, in millisecondi, prima che scada.
- `deep?: boolean`: indica se i dati ricevuti saranno reattivi con un livello di profondità maggiore di uno. `true` di default.
- `dedupe?: "cancel" | "defer"`: se `"cancel"`, le richieste duplicate vengono cancellate appena una nuova richiesta viene eseguita, se `"defer"` non vengono fatte nuove richieste. `"cancel"` di default.
- `pick?: string[]`: seleziona solo le proprietà specificate dell'oggetto ricevuto.
- `watch?: Ref[]`: un array di riferimenti reattivi. Al loro cambiamento scatta una nuova richiesta.
- `server?: boolean`: se `true`, la richiesta viene fatta lato server, altrimenti lato client. `true` di default.
- `lazy?: boolean`: se `true`, il rendering della pagina lato server è bloccante e aspetta che la richiesta sia completata prima di inviare la pagina al client. Se `false`, la richiesta viene fatta lato client, e la pagina viene inviata al client che aspetterà la risposta. `false` di default.
- `method?: "POST" | "GET"`: il metodo HTTP della richiesta
- `body?: any`: il corpo della richiesta, che può essere un oggetto, un array o una stringa
- `immediate: boolean`: se `true`, la richiesta viene fatta immediatamente al caricamento della pagina, altrimenti viene fatta solo quando il valore di `watch` cambia. `true` di default.

Inoltre è da notare che `useFetch` può essere usato con o senza `await`, a seconda se si vuole attendere la risposta della richiesta o meno. Se si usa `await`, il rendering di una pagina lato server aspetterà che la richiesta sia completata prima di inviare la pagina al client, mentre se non si usa `await`, la richiesta potrebbe eseguire in parallelo alla visualizzazione della pagina lato client, che quindi mostrerebbe una pagina incompleta. Per evitare problemi di SEO e LCP, è consigliato usare `await` in ogni caso.

Per ottenere dei comportamenti ad hoc per il caricamento di dati nella pagina, si possono combinare le opzioni `immediate`, `server` e `lazy`, come evidenzia il seguente[11] schema:

[11] Illustrato anche nel video tutorial di Matt Maribojoc.



Build per la produzione

Come mostrato nel paragrafo Command line interface, Nuxt dispone di un comando di build che permette di compilare il codice Typescript e di generare i file necessari per la distribuzione dell'applicazione. I file da inviare al client sono frammentati per migliorare le prestazioni di caricamento delle pagine, con una procedura chiamata *code splitting*. L'output della build dipende dalla modalità di rendering scelta.

2.1.4 Modalità di rendering del frontend

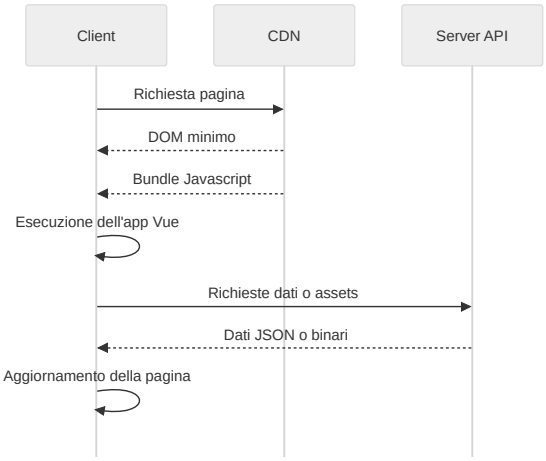
Tipi di applicazione diversi hanno esigenze diverse: Un blog o un sito vetrina potrebbero non richiedere le stesse prestazioni di un'applicazione di e-commerce o di un'applicazione di social networking. Nuxt si adatta a queste esigenze offrendo diverse modalità di rendering, che possono essere scelte per l'intera applicazione o per singole pagine o gruppi di pagine.

In questo contesto, con rendering di una pagina Web non si intende il processo di disegno dei pixel sullo schermo, del quale generalmente si occuperà il browser Web delegando al sistema operativo la gestione dell'hardware. Qui con rendering si intende il processo di generazione del codice HTML, CSS e Javascript che costituisce la pagina Web, a partire da componenti Vue, dati in oggetti Javascript e template.

Client Side Rendering

Nuxt supporta la stessa modalità di rendering discussa nel capitolo 1, in cui il codice dell'applicazione Vue viene eseguito interamente sul browser. Dopo una richiesta iniziale, il server invia un DOM minimo ed il bundle javascript al browser Web. Ogni richiesta successiva viene gestita dal client, che si occupa di fare le richieste al server API per ottenere i dati e di aggiornare il DOM

in base alle risposte. Gli `useFetch` che richiedono dati al caricamento iniziale della pagina vengono eseguiti sempre sul client.



Si può attivare globalmente nel file `nuxt.config.ts` con:

```
export default defineNuxtConfig({
    ssr: false,
});
```

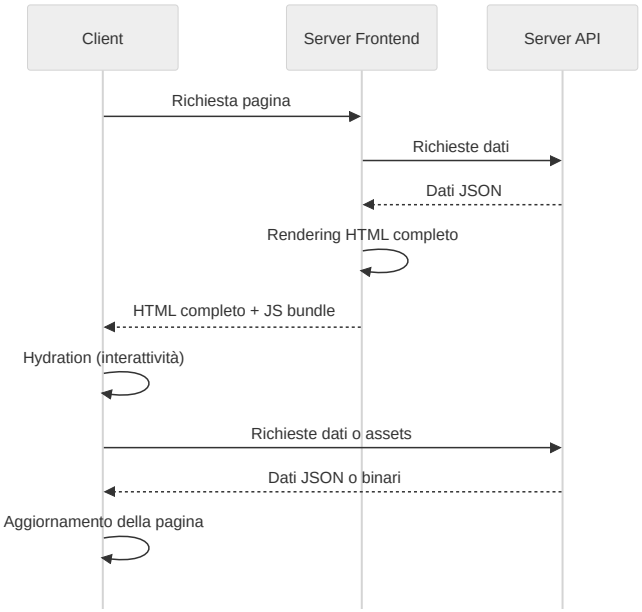
Il beneficio che si ottiene nello sviluppare in maniera CSR con Nuxt è una riduzione del costo infrastrutturale, perché il server backend non deve eseguire il codice Javascript per generare su richiesta la pagina: basterà infatti caricare il bundle dell'applicazione frontend generata con `nuxi build` su un server frontend statico [12] per distribuirla ad un numero indeterminatamente crescente di utenti. Tuttavia rimangono i problemi di performance, di accessibilità e di SEO che sono stati discussi precedentemente. Per i servizi API si possono usare *Functions as a Service*[13].

Server side rendering

Nuxt supporta anche un modello di rendering lato server, in cui, ad una richiesta iniziale, il codice dell'applicazione Vue viene eseguito per intero su un server frontend per inviare la pagina renderizzata come risposta. Il processo di Hydration aggiunge l'interattività dei componenti, che possono essere aggiornati, rimossi o aggiunti dinamicamente, senza dover ricaricare la pagina. Gli `useFetch` che richiedono dati al caricamento iniziale della pagina vengono eseguiti sul server. Poi quando il client riceve la pagina, può fare richieste API per ottenere dati aggiornati.

[12] Si tratta di un servizio di file statici, che può essere implementato anche con una *CDN* (Content Delivery Network) per distribuire i file in maniera efficiente in tutto il mondo.

[13] Secondo l'ISO/IEC 22123-2, una *function as a service* è un servizio cloud che esegue una funzione specifica su richiesta, senza la necessità per il programmatore di gestire l'infrastruttura sottostante.



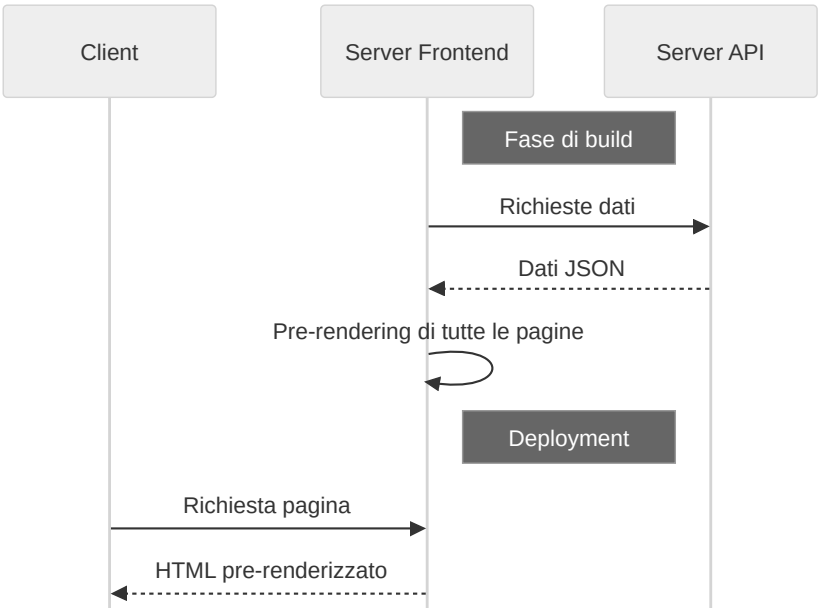
Si può attivare globalmente nel file `nuxt.config.ts` con:

```
export default defineNuxtConfig({
    ssr: true,
});
```

La SEO è migliorata perché i motori di ricerca possono leggere il codice HTML generato dal server frontend, e non devono aspettare che il codice Javascript venga eseguito sul browser. Le prestazioni di SSG sono le più soddisfacenti per una esperienza utente generale, tuttavia, il costo infrastrutturale è maggiore: il server frontend è sempre sotto elevati carichi di lavoro, e deve essere dimensionato di conseguenza.

Static site generation

Nuxt supporta la generazione di siti statici, cioè la generazione di pagine HTML in fase di build. Non avviene nessun rendering lato server durante la fase di produzione, ma solo in fase di pubblicazione. Questo modello è adatto per siti Web che non richiedono interattività o aggiornamenti frequenti, come blog, documentazioni o siti vetrina. Gli `useFetch` che richiedono dati al caricamento iniziale della pagina vengono eseguiti durante la fase di build.



Si può attivare globalmente nel file `nuxt.config.ts` con:

```
export default defineNuxtConfig({
    ssr: true,
    nitro: {
        preset: "static",
    },
});
```

Per poi generare il sito statico con:

```
npx nuxi generate
```

Le prestazioni sono le migliori possibili, perché il server frontend non deve eseguire il codice Javascript per generare la pagina, ma solo inviare il file HTML pre-renderizzato. Anche la SEO è nelle condizioni più favorevoli, perché i motori di ricerca leggeranno lo stesso HTMl inviato ai client degli utenti. Tuttavia, l'interattività è limitata in quanto ogni componente Vue che viene reso dinamico dopo l'idratazione soffrirà degli stessi problemi di un'applicazione CSR, in scala ridotta. Per questo motivo si sceglie SSG quando non c'è necessità di gestire richieste API con alcun tipo di server backend.

Incremental static regeneration

È diffuso un modello simile al SSG, chiamato *incremental static regeneration*, in cui le pagine statiche vengono rigenerate in base a un intervallo di tempo o a un evento specifico. È quindi necessario un server frontend con capacità di calcolo modeste, in grado di fare periodicamente richieste al server backend per potersi aggiornare. Questo modello è adatto per siti Web che richiedono aggiornamenti periodici, come siti che forniscono un feed di notizie altamente personalizzato o siti di

e-commerce che devono fornire prezzi dei prodotti aggiornati al first contentful paint. Come per la SSG, gli `useFetch` che richiedono dati al caricamento iniziale della pagina vengono eseguiti durante la fase di aggiornamento della cache.

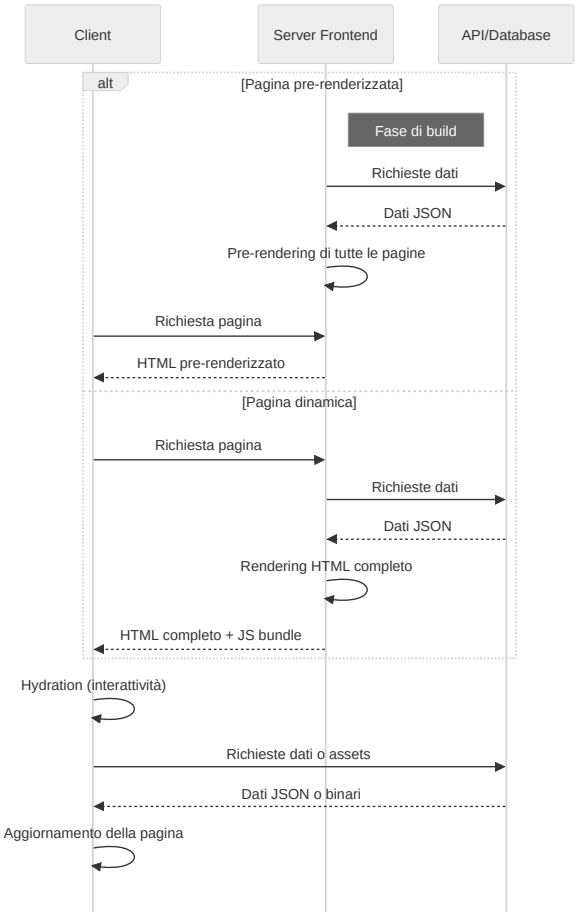
```
    participant client as Client
    participant frontend as Server Frontend
    participant backend as API/Database
    client->>frontend: Richiesta pagina
    alt Cache valida
        frontend-->>client: HTML pre-renderizzato dalla cache
    else Cache scaduta o non esistente
        frontend->>backend: Richieste dati
        backend-->>frontend: Dati JSON
        frontend->>frontend: Rendering HTML
        frontend->>frontend: Aggiornamento cache
        frontend-->>client: HTML appena generato
    end
    client->>client: Hydration (interattività)

    client->>backend: Richieste dati o assets
    backend-->>client: Dati JSON o binari
    client->>client: Aggiornamento della pagina
```

Si può scegliere l'intervallo di invalidazione per certe pagine nel file `nuxt.config.ts`. Con `swr` si configura l'header HTTP *stale-while-revalidate*, indicando il tempo in secondi per cui la cache è considerata valida. Il client quindi è a conoscenza di quando il server frontend genererà una nuova versione della pagina.

```
export default defineNuxtConfig({
    ssr: true,
    routeRules: {
        "/blog/today/**": {
            swr: 60,
        },
        "/blog/this-year/**": {
            swr: 3600,
        },
    },
});
```

Universal rendering

Infine Nuxt supporta un modello di rendering ibrido tra static site generation e server side rendering.



Si può attivare per insiemi di rotte nel file `nuxt.config.ts` con:

```
export default defineNuxtConfig({
    routeRules: {
        "/about": { prerender: true },
        "/blog/**": { prerender: true },
        "/dashboard": { ssr: true },
    },
});
```

In questo esempio le pagine `about` e tutte le pagine di `blog/` vengono pre-renderizzate, cioè generate in fase di build, mentre la pagina `dashboard` viene renderizzata lato server, cioè generata al momento della richiesta, in modo da combinare il vantaggio di costo ridotto ed elevate prestazioni della SSG con l'interattività e la personalizzazione della SSR.

2.2 TypeORM

TypeORM è una libreria per ORM (Object-Relational Mapping) basata su Typescript, che permette di rappresentare le entità e le relazioni di un database relazionale in modo dichiarativo, e di eseguire operazioni *CRUD* (Create, Read, Update, Delete) su di esse con API type-safe.

Il progetto, avviato nel 2016 da Umed Khudoiberdiev, è attualmente mantenuto da un team di sviluppatori che accettano contributi, all'indirizzo `github.com/typeorm/typeorm`. La versione stabile corrente è la **0.3.20**, rilasciata nel gennaio 2024, ma lo sviluppo di versioni *nightly* è in corso. Attualmente TypeORM è usato come dipendenza in quasi 400'000 progetti su Github.

Si può installare in un progetto Node con `npm install typeorm`, e richiede `typescript` con versione 4.5 o successiva, con le dichiarazioni di tipo `@types/node`. L'uso di TypeORM come libreria in un tag script di un file HTML non è supportato.

2.2.1 Command line interface

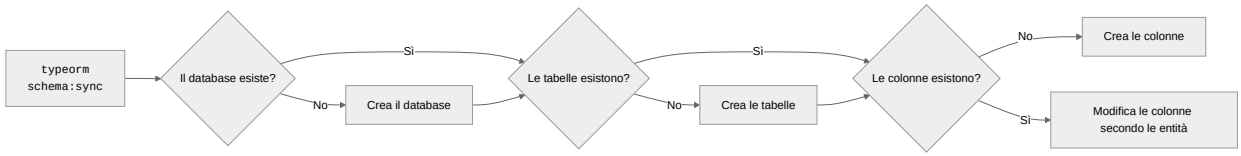
Si può avviare un progetto con la CLI di TypeORM con `npx typeorm init --database <database>`, scegliendo tra i seguenti database: `mysql`, `mariadb`, `postgres`, `cockroachdb`, `sqlite`, `mssql`, `sap`, `spanner`, `oracle`, `mongodb`, `cordova`, `react-native`, `expo`, `nativescript`. Una volta che il progetto è configurato si possono eseguire i seguenti comandi:

`typeorm entity:create <percorso>`

Genera il file di una nuova entità in una directory specificata. Si tratta di un file Typescript che rappresenta una tabella del database, con i campi e le relazioni definite come proprietà della classe.

`typeorm schema:sync`

Sincronizza il database con le entità definite nel progetto. Il protocollo di sincronizzazione è evidenziato nel diagramma di flusso sotto.



`typeorm schema:drop`

Elimina tutte le tabelle del database.

`typeorm schema:log`

Stampa su *stdout* le query SQL che verranno eseguite dal comando `schema:sync`.

`typeorm query <query>`

Esegue una query SQL sul database, nel dialetto del DBMS specificato.

`typeorm cache:clear`

Svuota la cache delle query.

`typeorm subscriber:create <percorso>`

Crea un nuovo subscriber, cioè una funzione che viene eseguita quando si verifica un evento sul database.

`typeorm migration:create <percorso>`

Crea un nuovo file di migrazione, che potrà essere utilizzato per sincronizzare il database successivamente.

`typeorm migration:run`

Esegue tutte le migrazioni pendenti, cioè le modifiche allo schema del database che non sono state ancora applicate.

`typeorm migration:show`

Stampa su stdout le migrazioni pendenti.

`typeorm migration:revert`

Annulla l'ultima migrazione eseguita.

`typeorm migration:generate <percorso>`

Genera una migration a partire dalle differenze tra le entità e le tabelle del database.

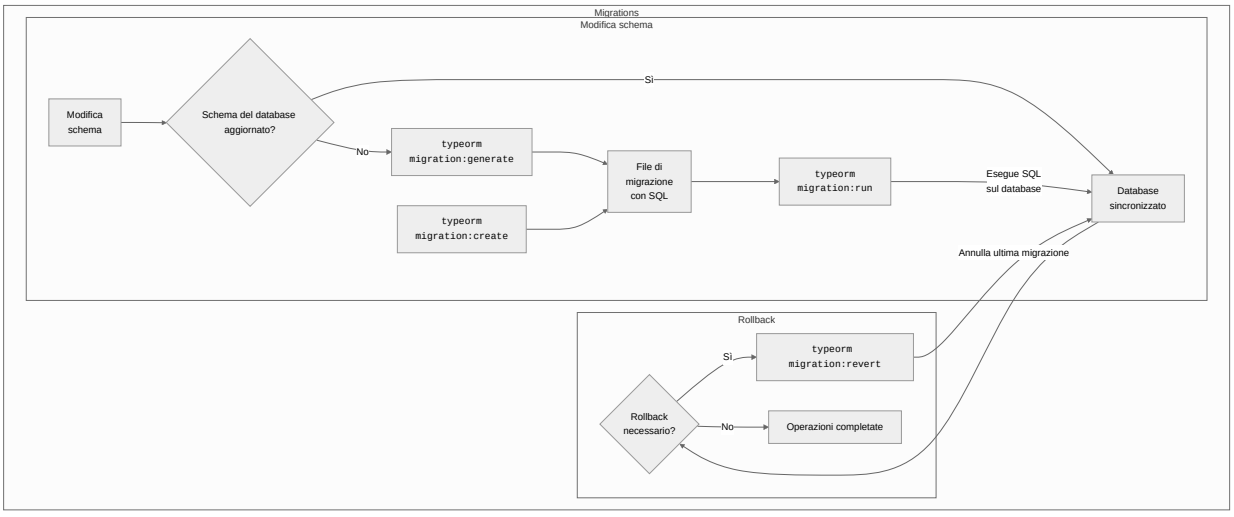


Diagramma di flusso delle migrazioni in TypeORM. L'esecuzione del SQL è specifica per il DBMS scelto, e può essere differente tra i vari database supportati.

2.2.2 Collegamento con il database

TypeORM consente di lavorare con diversi DBMS (Database Management Systems), tra cui:

DBMS:	Relazionale?	Server based?	Adattatore:
MySQL o MariaDB	✅	✅	`mysql2`
PostgreSQL	✅	✅	`pg`
SQLite	✅	su file	`sqlite3`
Sql.js	✅	in memoria	`sql.js`
Microsoft SQL Server	✅	✅	`mssql`
OracleDB	✅	✅	`oracledb`
MongoDB[14]	❌, a documenti	✅	`mongodb`
SAP Hana	✅	in memoria	`hdb-pool`
Google Cloud Spanner	✅	✅	`spanner`

Per effettuare la connessione con il database occorre installare l'adattatore mediante npm, e configurare un istanza di `DataSource` con le opzioni di accesso al database.

Ad esempio, in PostgreSQL:

```
import { DataSource } from "typeorm";
import type { DataSourceOptions } from "typeorm";

let options: DataSourceOptions = {
    type: "postgres",
    host: "localhost",
    port: 5432,
    database: "dev",
    username: "dev",
    password: "dev",
    ssl: false,
    connectTimeoutMS: 10000,
    synchronize: true,
    logging: true,
    entities: [],
    migrations: [],
    subscribers: [],
};

const AppDataSource = new DataSource(options);
```

Le opzioni di connessione[15] sono specifiche per ogni adattatore, ma quelle comuni sono:

[14] MongoDB è un database NoSQL, ma TypeORM supporta la connessione ad esso ed un'API simile a quella che include per i database relazionali.

[15] È possibile consultare quali opzioni sono disponibili per i vari adattatori qui.

- `type: string`: il tipo di database, tra quelli supportati da TypeORM. In base a questo campo il compilatore Typescript inferirà il tipo specifico delle opzioni.
- `entities: EntitySchema[]`: un array di classi che rappresentano le migrazioni del database
- `migrations: Function[]`: un array di classi che rappresentano le migrazioni del database.
- `subscribers: Function[]`: un array di classi che rappresentano i subscriber del database.
- `synchronize: boolean`: indica se sincronizzare il database con le entità definite nel progetto.
- `logging: boolean | ["query", "error", "schema", "warn", "info", "log"] | AbstractLogger`: abilita la stampa su stdout delle query SQL eseguite sul database durante l'esecuzione dell'applicazione e permette di specificare quali tipi di log abilitare o di passare un logger personalizzato.
- `cache: boolean | {type: ["database", "redis", ...],options: {...} }`: abilita la cache delle query, con la possibilità di specificare il tipo di cache, tra le quali anche Redis.

Per i DBMS che richiedono una connessione a server, si aggiungono le opzioni:

- `host: string`: l'indirizzo IP o il nome del server.
- `port: number`: la porta del server.
- `database: string`: il nome del database.
- `username: string`: l'utente del database.
- `password: string`: la password dell'utente.
- `ssl: boolean`: abilita la connessione sicura.

In più, per PostgreSQL, si può specificare:

- `connectTimeoutMS: number`: il tempo massimo di attesa per la connessione al server, in millisecondi.
- `uuidExtension: boolean`: abilita l'estensione UUID di PostgreSQL.

In un'applicazione Typescript che viene impacchettata per l'utilizzo su client (browser) si può usare il modulo `typeorm/browser` per interagire con `indexedDB`, un database locale che è supportato da tutti i browser moderni, per memorizzare i dati in locale.

```
import { DataSource } from "typeorm/browser";

const AppDataSource = new DataSource({
    type: "indexeddb",
    database: "mydb",
    entities: [],
    synchronize: true,
});
```

Successivamente si può inizializzare la connessione con il database con il metodo `initialize`, per iniziare a fare queries.

```
export async function initialize() {
    try {
```

```
        if (!AppDataSource.isInitialized) {
            await AppDataSource.initialize();
            console.log("Typeorm inizializzato");
        }
    } catch (error) {
        console.error("Errore inizializzazione Typeorm", error);
        throw error;
    }
}
```

Questa operazione è asincrona, quindi si può usare `await` per attendere il completamento dell'inizializzazione, ed è da preferire eseguirla all'avvio dell'applicazione, per evitare errori di connessione al database durante l'esecuzione.

2.2.3 Rappresentazione di entità

Entità

Una delle features principali di TypeORM è la possibilità di definire le entità del database come classi Typescript con *decoratori*[16], al contrario di altri ORM che usano un formato di configurazione esterno, come Prisma, o che usano un formato di configurazione interno, come Sequelize.

È necessario installare uno *shim* (una libreria che si interpone tra due API) per poter usare i decoratori di TypeORM, con `npm install reflect-metadata --save`.

È inoltre necessario configurare il compilatore typescript per supportare i decoratori, aggiungendo le seguenti opzioni al file `tsconfig.json`:

```
{
    "compilerOptions": {
        "experimentalDecorators": true,
        "emitDecoratorMetadata": true
    }
}
```

Poi bisogna importare il modulo `reflect-metadata` globalmente in un file di entry del progetto:

```
import "reflect-metadata";
```

Si potrà quindi definire la tabella `user`[17] con la classe `User`:

```
export class User {
    public id: number;
    public firstName: string;
    public lastName: string;
    public fullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

[16] I decoratori sono funzioni che modificano il comportamento di una classe o di una funzione, aggiungendo o modificando proprietà o metodi. Sono stati introdotti in ES7 e sono supportati da Typescript.

[17] A riguardo, il paragrafo Strategie di naming automatico.

Poi si annota la classe con il decoratore `@Entity`:

```
import { Entity } from "typeorm";
@Entity()
export class User {
    // ...
}
```

E successivamente si annotano i campi che si intende tradurre in colonne della tabella con il decoratore `@Column`:

```
import { Entity, Column } from "typeorm";
@Entity()
export class User {
    @Column()
    public id: number;
    @Column()
    public firstName: string;
    @Column()
    public lastName: string;
    public fullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

In questo modo il tipo di dato da assegnare alla colonna è inferito automaticamente dal tipo della proprietà, e si può continuare ad usare il metodo `fullName` per ottenere il nome completo dell'utente.

Per rendere le entità persistenti si devono aggiungere i riferimenti delle classi ad un array nel campo `entities` delle opzioni di `DataSource`.

Entità annidate

Un'entità può supportare delle entità annidate, che sono rappresentate come proprietà di tipo `Entity`:

```
import { Name } from "./Name";

@Entity()
export class User {
    @Column()
    id: number;

    @Column(() => Name)
    name: Name;
}
```

Dove `Name` è una classe che rappresenta un nome, con delle `@Column` sui campi da annidare.

```ts
export class Name {
    @Column()
    first: string;

    @Column()
    last: string;
}
```

In questo modo la tabella risultante sarà:

user	
number	id
varchar	name_first
varchar	name_last

Quindi si potrà accedere ai campi annidati con `user.name.first` e `user.name.last`, e per ogni entità che fa uso di `Name` si otterrà una riduzione della ridondanza del codice.

Polimorfismo delle entità

Le entità possono essere definite in modo polimorfico, cioè con una gerarchia di classi che condividono delle proprietà comuni.

Si inizia definendo un'entità padre `User`:

```ts
@Entity()
export class User {
    @Column()
    id: number;

    @Column()
    firstName: string;

    @Column()
    lastName: string;
}
```

Poi si definiscono le entità figlie `Admin` e `Customer`:

```ts
@Entity()
export class Supplier extends User {
    @Column()
    companyName: string;
}
```

```typescript
@Entity()
export class Customer extends User {
    @Column()
    shippingAddress: string;
}
```

Il diagramma delle tabelle risultante sarà:

user	
number	id
varchar	firstName
varchar	lastName

supplier	
number	id
varchar	firstName
varchar	lastName
varchar	companyName

customer	
number	id
varchar	firstName
varchar	lastName
varchar	shippingAddress

L'ereditarietà è supportata ad un livello di profondità arbitrario.

È possibile usare anche il pattern di *Single table inheritance* (STI), in cui tutte le entità figlie condividono la stessa tabella, e si usa un campo `type` per distinguere i diversi tipi di entità.

```typescript
@Entity()
@TableInheritance({ column: { type: "varchar", name: "type" } })
export class User {
    ...
}

export class Supplier extends User {
    ...
}

export class Customer extends User {
    ...
}
```

Così facendo si otterrà un'unica tabella `user` con un campo `type` che può assumere i valori `Supplier` e `Customer`:

user	
number	id
varchar	firstName
varchar	lastName
varchar	type
varchar	companyName
varchar	shippingAddress

Schemi

È anche possibile definire uno schema per le tabelle, senza l'utilizzo di decoratori, facendo utilizzo di interfacce Typescript e di classi EntitySchema con un parametro *generic* dell'interfaccia in considerazione. l'esempio del paragrafo entità annidate può essere riscritto come:

```ts
export interface Name {
    first: string;
    last: string;
}

export const NameEntitySchema = new EntitySchema<Name>({
    name: "name",
    columns: {
        first: { type: "varchar" },
        last: { type: "varchar" },
    },
});

export interface User {
    id: string;
    name: Name;
}

export const UserEntitySchema = new EntitySchema<User>({
    name: "user",
    columns: {
        id: { primary: true, type: "int", generated: true },
    },
    embeddeds: {
        name: { schema: NameEntitySchema, prefix: "name_" },
    },
});
```

Si passa un oggetto di tipo `EntitySchemaOptions` al costruttore di `EntitySchema`, che contiene le opzioni di configurazione della tabella, tra cui:

- `name: string`: il nome della tabella.
- `columns: EntitySchemaOptions<T>.columns`: un oggetto che mappa i nomi delle colonne, tipizzati come string literals, ad ulteriori opzioni della colonna, cioè ad un `EntitySchemaColumnOptions`. Ognuno di questi ha i campi:
  - `type: ColumnType`: il tipo di dato della colonna, istanza di `ColumnType`.
  - `primary?: boolean`: se la colonna è parte della chiave primaria, `false` di default.
  - `generated?: boolean`: se il valore della colonna è generato automaticamente, `false` di default.
  - `default?: any`: il valore di default della colonna.
- `embeddeds: EntitySchemaColumnOptions[]`: un oggetto che mappa i nomi delle proprietà annidate ai loro schemi e ai prefissi delle colonne annidate. Per ogni proprietà annidata si deve definire:
  - `schema: EntitySchema`: lo schema dell'entità annidata.
  - `prefix: string`: il prefisso delle colonne annidate.

La tipizzazione forte è garantita anche nel caso degli schemi.

Proprietà delle colonne

Nelle `@Entity` si possono impostare proprietà delle colonne del database sempre mediante decoratori:

- `@PrimaryGeneratedColumn()`: Chiave primaria generata automaticamente.
- `@PrimaryColumn()`: Chiave primaria.
- `@Column("int")`: Tipo di dato della colonna. Un `number` Typescript può essere mappato a `int` in SQL.
- `@Column("varchar")` oppure `@Column("text")`: Tipo di dato della colonna. Una `string` Typescript può essere mappata a `varchar` o `text` in SQL.

Si possono aggiungere delle proprietà aggiuntive passando un oggetto di opzioni, di tipo `ColumnOptions`, al decoratore `@Column(options: ColumnOptions)`:

Alcune delle opzioni più comuni per PostgreSQL sono:

- `type: ColumnType`: Il tipo di dato della colonna, istanza di `ColumnType`, tra i quali:
  - `"int"`: Un intero a 32 bit.
  - `"bigint"`: Un intero a 64 bit.
  - `"varchar"`: Una stringa di lunghezza variabile.
  - `"text"`: Una stringa di lunghezza arbitraria.
  - `"boolean"`: Un valore booleano.
  - `"date"`: Una data senza orario, in TypeScript un oggetto `Date`.
  - `"timestamp"`: Una data e un orario, in TypeScript un oggetto `Date`.
  - `"json"`: Un oggetto JSON. Con questo tipo si possono memorizzare oggetti complessi, come array e oggetti annidati. L'utilizzo di dati non strutturati come JSON può estendere le capacità di database relazionali che li supportano ed avvicinarli a NoSQL[18], ma può rendere più complesse le query e meno efficienti le operazioni di ricerca e ordinamento. Al momento di scrittura, TypeORM non offre supporto per l'API `find` (vedi in seguito) per le sotto-strutture di una colonna JSON[19].
  - `"jsonb"`: Un oggetto JSON binario.
  - `"enum"`: Un insieme di valori possibili. Si definisce con un array di stringhe, che viene tipizzato come uno *string literal type*, o anche con un enum Typescript. In entrambi i casi la tipizzazione è garantita.
- `length: number`: La lunghezza massima della colonna, per i tipi `varchar` e `text`.
- `nullable: boolean`: Se la colonna può avere valori nulli.
- `default: any`: Il valore di default della colonna.
- `unique: boolean`: Se i valori della colonna devono essere unici.
- `primary: boolean`: Se la colonna è parte della chiave primaria.
- `generated: boolean`: Se il valore della colonna è generato automaticamente.
- `comment: string`: Un commento sulla colonna.

[18] Che cos'è NoSQL - Articolo sul blog di MongoDB.
[19] Issue 11016 di TypeORM.

Migrations

Infine, per modificare lo schema del database, una volta che le entità sono definite, si possono creare delle migrazioni, che sono file Typescript che contengono le query SQL necessarie per modificare lo schema del database.

```
import { MigrationInterface, QueryRunner } from "typeorm";

export class PostRefactoringTIMESTAMP implements MigrationInterface {
    async up(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.query(
            `ALTER TABLE "user" RENAME COLUMN "lastname" TO "surname"`
        );
    }
    async down(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.query(
            `ALTER TABLE "user" RENAME COLUMN "surname" TO "lastname"`
        );
    }
}
```

È anche disponibile l'API `QueryRunner`, che permette di astrarre parti di sintassi SQL specifiche per i vari DBMS, ma i parametri delle query non sono tipizzati.

```
import { MigrationInterface, QueryRunner, TableColumn } from "typeorm";

export class PostRefactoringTIMESTAMP implements MigrationInterface {
    async up(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.renameColumn("user", "lastname", "surname");
    }

    async down(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.renameColumn("post", "surname", "lastname");
    }
}
```

La pratica di migrazione di database è tuttavia consigliata per modifiche in fase di produzione, perché aggiornare le entità di TypeORM e sincronizzare il database può portare a perdita di dati.

2.2.4 Rappresentazione di relazioni

Dopo la definizione delle entità, si possono definire le relazioni tra di esse, seguendo le convenzioni del modello relazionale.

Si possono definire relazioni uno a uno con `@OneToOne`, relazioni uno a molti con `@OneToMany` e `@ManyToOne`, e relazioni molti a molti con `@ManyToMany`. Ogni decoratore per le relazioni, accetta un argomento `typeFunctionOrTarget` che specifica il tipo di entità correlata con una funzione che restituisce il `type`. Opzionalmente accetta un argomento `inverseSide`, che specifica la proprietà della relazione inversa, tramite una funzione che prende come argomento il tipo di entità corrente

e restituisce il suo `field` che inverte la relazione. Si passa infine un oggetto `RelationOptions`, che permette di configurare la relazione con:

- `cascade?: boolean | ("insert" | "update" | "remove" | "soft-remove" | "recover")[]`: specifica le operazioni che devono essere propagate alla relazione. Se `true`, tutte le operazioni di `insert`, `update` e `remove` sono propagate. Se `false`, nessuna operazione è propagata. Se un array di stringhe tra quelle accettabili, solo le operazioni specificate sono propagate. `false` di default.
- `eager?: boolean`: se il caricamento della relazione deve essere eager, `true` di default.
- `lazy?: boolean`: se il caricamento della relazione deve essere lazy.
- `orphanedRowAction?: "nullify" | "delete" | "soft-delete" | "disable"`: specifica l'azione da intraprendere quando un'entità correlata è rimossa. Se `nullify`, la chiave esterna è impostata a `null`. Se `delete`, l'entità correlata è rimossa. Se `soft-delete`, l'entità correlata è marcata come eliminata. Se `disable`, l'entità correlata è marcata come disabilitata. `nullify` di default.

Relazioni uno a uno

Nell'esempio che segue, ogni utente ha un solo profilo, e ogni profilo è associato ad un solo utente. È impostato il `cascade` a `true`, in modo che le operazioni di `insert`, `update` e `remove` siano propagate solo nella direzione `User -> Profile`. Questo prevede che un profilo venga creato, aggiornato o rimosso solo quando un utente è già presente, e che un utente venga rimosso solo se il profilo è già stato rimosso.

È fatto uso di una colonna di join, `profile_id`, e punta alla tabella `profiles`. Si usa il decoratore `@JoinColumn()`, che accetta un oggetto di tipo `JoinColumnOptions` per specificare le opzioni della colonna di join, tra cui:

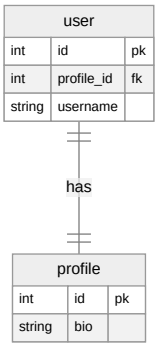
- `name: string`: il nome della colonna di join.
- `referencedColumnName: string`: il nome della colonna di riferimento.
- `foreignKeyConstraintName: string`: il nome del vincolo di chiave esterna.

Questi parametri sono opzionali perché TypeORM usa delle convenzioni per inferire i nomi delle tabelle e delle colonne[20].

```
1  @Entity()
2  export class User {
3      @PrimaryGeneratedColumn()
4      id: number;
5
6      @Column()
7      username: string;
8
9      @OneToOne(() => Profile, (profile) => profile.user, {
10         cascade: true,
```

[20] A riguardo, il paragrafo Strategie di naming automatico.

```
    })
    @JoinColumn()
    profile: Profile;
}

@Entity()
export class Profile {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    bio: string;

    @OneToOne(() => User, (user) => user.profile)
    user: User;
}
```

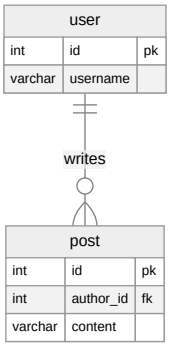


Relazioni uno a molti e molti a uno

Nell'esempio che segue, ogni utente può scrivere molti post, e ogni post è scritto da un solo utente. Dunque la tabella `posts` ha una chiave esterna `author_id` che punta alla tabella `users`.

```
@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    username: string;

    @OneToMany(() => Post, (post) => post.author)
    posts: Post[];
}

@Entity()
export class Post {
    @PrimaryGeneratedColumn()
```

```
16      id: number;
17
18      @Column()
19      content: string;
20
21      @ManyToOne(() => User, (user) => user.posts)
22      @JoinColumn()
23      author: User;
24  }
```



Relazioni molti a molti

Nell'esempio che segue ogni post è scritto da un utente. Ogni utente può mettere "mi piace" a molti post, e ogni post può ricevere il "mi piace" da molti utenti. Viene generata una tabella di join `user_liked_posts`[21] ha due chiavi esterne, `user_id` e `post_id`, che puntano rispettivamente alle tabelle `users` e `posts`. La coppia di chiavi `user_id` e `post_id` è unica, quindi forma una chiave primaria composta per la tabella `user_liked_posts`.

In più ogni utente può avere molti amici, che sono altri utenti, quindi viene generata una tabella di join `user_friends` con due chiavi esterne, `user1_id` e `user2_id`, che puntano entrambe alla tabella `users`. La coppia di chiavi `user1_id` e `user2_id` è unica, quindi forma una chiave primaria composta per la tabella `user_friends`.

Ogni utente può seguire altri utenti, e può essere seguito da altri utenti. Quindi viene generata la tabella di join, `user_following`, con due chiavi esterne, `follower_id` e `following_id`, che puntano entrambe alla tabella `users`. La coppia di chiavi `follower_id` e `following_id` è unica, quindi forma una chiave primaria composta per la tabella `user_following`. Da questa tabella si può ottenere con efficienza sia l'elenco degli utenti seguiti da un utente, sia l'elenco degli utenti che seguono un utente.

È fatto uso di `@JoinTable()` per specificare il nome della tabella di join e le colonne che la compongono. È passato un oggetto di tipo `JoinTableOptions`, che permette di configurare la tabella di join con:

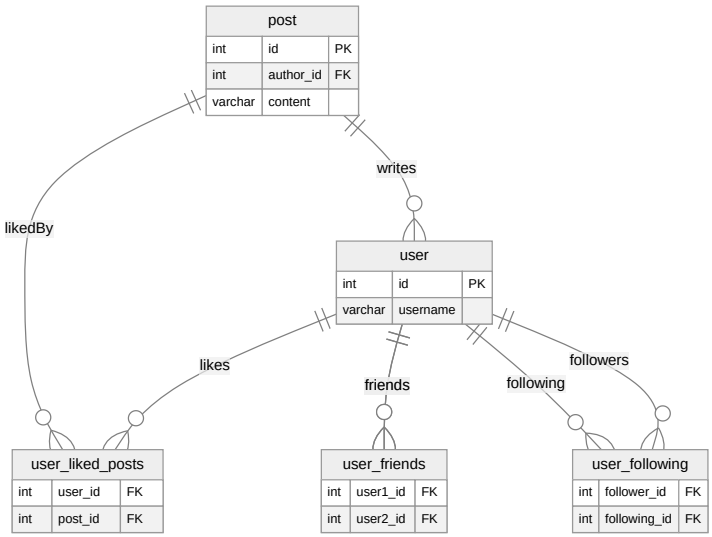
- `name: string`: il nome della tabella di join.
- `joinColumn: JoinColumnOptions`: le opzioni per la colonna di join.

[21] A riguardo, il paragrafo Strategie di naming automatico.

- `inverseJoinColumn: JoinColumnOptions`: le opzioni per la colonna di join inversa, con gli stessi campi di `joinColumn`.
- `database: string`: il nome del database in cui creare la tabella di join.
- `synchronize: boolean`: se sincronizzare la tabella di join con il database, `true` di default.

```
@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    username: string;

    @OneToMany(() => Post, (post) => post.author)
    posts: Post[];

    @ManyToMany(() => Post, (post) => post.likedBy)
    likedPosts: Post[];

    @ManyToMany(() => User, (user) => user.friends)
    @JoinTable({
        name: "user_friends",
        joinColumn: {
            name: "user1_id",
            referencedColumnName: "id",
        },
        inverseJoinColumn: {
            name: "user2_id",
            referencedColumnName: "id",
        },
    })
    friends: User[];

    @ManyToMany(() => User, (user) => user.followers)
    @JoinTable({
        name: "user_following",
        joinColumn: {
            name: "follower_id",
            referencedColumnName: "id",
        },
        inverseJoinColumn: {
            name: "following_id",
            referencedColumnName: "id",
        },
    })
    following: User[];

    @ManyToMany(() => User, (user) => user.following)
    followers: User[];
```

```
}

@Entity()
export class Post {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    content: string;

    @ManyToOne(() => User, (user) => user.posts)
    @JoinColumn()
    author: User;

    @ManyToMany(() => User, (user) => user.likedPosts)
    @JoinTable()
    likedBy: User[];
}
```



Eager e lazy loading

Il caricamento delle relazioni può essere configurato come *eager* o *lazy*.

Con un caricamento eager le entità correlate vengono caricate insieme all'entità principale, in un'unica query SQL. Questo può essere utile quando si prevede che le entità correlate verranno sempre usate insieme all'entità principale, ma può portare ad inefficienze se queste sono molte o pesanti.

```
@Entity()
export class User {
    @ManyToMany(() => Post, (post) => post.likedBy, { eager: true })
    @JoinTable()
```

```
    likedPosts: Post[];
}

@Entity()
export class Post {
    @ManyToMany(() => User, (user) => user.likedPosts, { eager: true })
    likedBy: User[];
}
```

Al contrario, con un caricamento lazy le entità correlate vengono caricate solo quando vengono effettivamente usate. Le entità correlate sono memorizzate come `Promise`, che se risolte restituiscono l'entità correlata. Durante la risoluzione della `Promise` viene eseguita una query SQL per recuperare l'entità correlata.

```
@Entity()
export class User {
    @ManyToMany(() => Post, (post) => post.likedBy, { lazy: true })
    @JoinTable()
    likedPosts: Promise<Post[]>;
}

@Entity()
export class Post {
    @ManyToMany(() => User, (user) => user.likedPosts, { lazy: true })
    likedBy: Promise<User[]>;
}
```

Internamente TypeORM usa delle `Proxy` Javascript per intercettare gli accessi alle proprietà lazy, in modo concettualmente analogo a:

```
user.likedPosts = new Proxy(Promise.resolve([]), {
    get(target, prop) {
        if (!target.__loaded) {
            target.__loaded = true;
            target.__data = databaseQuery(
                "SELECT * FROM post WHERE userId = ?",
                user.id
            );
        }
        return Reflect.get(target.__data, prop);
    },
});
```

Strategie di naming automatico

Per le definizioni sopra riportate, i nomi delle tabelle e delle colonne possono essere inferiti da TypeORM, seguendo delle convenzioni di naming automatico. Queste convenzioni possono essere sovrascritte con delle opzioni specifiche, passate ai decoratori delle entità e delle colonne, ma di default sono:

- Il nome delle tabella è il nome della classe, in minuscolo e con gli spazi sostituiti da `_`.
- Il nome delle colonne è il nome della proprietà, in minuscolo e con gli spazi sostituiti da `_`.
- Il nome di tabelle di join è il nome delle entità coinvolte, in ordine `primario_secondario`, in minuscolo e con gli spazi sostituiti da `_`.
- Il nome delle chiavi esterne è il nome della tabella di riferimento, in minuscolo e con gli spazi sostituiti da `_`, seguito dal nome del suo campo chiave.

Si può assegnare un'oggetto di classe che implementa `NamingStrategyInterface` al campo `namingStrategy` di `DataSource`, come ad esempio:

```
import { NamingStrategyInterface, DefaultNamingStrategy } from "typeorm";
import { snakeCase } from "typeorm/util/StringUtils";

export class CustomNamingStrategy
    extends DefaultNamingStrategy
    implements NamingStrategyInterface
{
    tableName(
        targetName: string,
        userSpecifiedName: string | undefined
    ): string {
        return userSpecifiedName || snakeCase(targetName);
    }

    columnName(
        propertyName: string,
        customName: string,
        embeddedPrefixes: string[]
    ): string {
        return customName || snakeCase(propertyName);
    }
}
```

Listeners e subscribers

È fornita un'API per definire *listeners* e *subscribers* per le entità, che permettono di eseguire del codice in risposta a eventi specifici, come il caricamento di un'entità dal database.

Listeners Si può definire un listener che ascolta un'evento su un'entità specifica, usando i decoratori `@BeforeLoad`, `@AfterLoad`, `@BeforeInsert`, `@AfterInsert`, `@BeforeUpdate`, `@AfterUpdate`, `@BeforeRemove`, `@AfterRemove`. Ad esempio:

```
@Entity()
export class Post {
    @AfterLoad() {
        console.log("Post con contenuto: ", this.content, " caricato");
    }

```

```
    @Column()
    content: string;
}
```

Tuttavia non si possono eseguire query a database senza garanzia di risoluzione di corse critiche. Per ovviare a questo problema si possono usare i subscribers.

Subscribers Un subscriber ha la stessa funzione di un listener, ma è definito come una classe che implementa l'interfaccia `EntitySubscriberInterface<Entity>`. Si possono definire metodi per gestire gli eventi di un'entità specifica, e si può fare query a database in modo asincrono. Ad esempio:

```
@EventSubscriber()
export class PostSubscriber implements EntitySubscriberInterface<Post> {
    listenTo() {
        return Post;
    }

    afterLoad(event: LoadEvent<Post>) {
        console.log("Post con contenuto: ", event.entity.content, " caricato");
        // query a database ...
    }
}
```

2.2.5 Query

TypeORM fornisce diverse API per eseguire query CRUD , oltre a supportare transazioni ACID. Typescript garantisce la type safety fino a momento di compilazione: è possibile scrivere metodi ed interfacce che usano le classi delle entità per garantire che gli inserimenti da parte dell'utente siano corretti a tempo di esecuzione. TypeORM fornisce un sistema di *sanitization* dei dati, ma la responsabilità di garantire la correttezza dei dati, quando ad esempio si tratta di validare un indirizzo email, rimane a carico dello sviluppatore.

Ad ogni entità allegata al DataSource è associato un *repository*, che permette di eseguire query. Il risultato di queste è incapsulato in una `Promise`, che può essere risolta con `await` per ottenere il risultato effettivo. Ogni transazione garantisce che tutte le operazioni al suo interno vengano eseguite con successo o nessuna di esse venga applicata, evitando stati inconsistenti del database. L'isolamento da corse di lettura e scrittura è garantito dalle transazioni, anche risolvendo parallelamente le promise con `Promise.all()`.

Sono supportati due pattern principali: Active Record per eseguire query CRUD direttamente sulle entità, e Query Builder per costruire query SQL in modo programmatico. È inoltre disponibile un API per eseguire query SQL direttamente.

Active record

Il pattern Active Record, per la prima volta introdotto da Ruby on Rails, permette di eseguire operazioni CRUD in modo coerente al paradigma ad oggetti. In TypeORM ogni classe `Entity`

che rappresenta un'entità del database che estende `BaseEntity` ed è associata staticamente ad un oggetto `DataSource`, ha a disposizione i seguenti:

Metodi statici:

- `getRepository(): Repository<Entity>`: restituisce il repository dell'entità.
- `find(conditions?: FindManyOptions<Entity>): Promise<Entity[]>`: trova tutte le entità che soddisfano le condizioni specificate.
- `findOne(conditions?: FindOneOptions<Entity>): Promise<Entity>`: trova tutte le entità che soddisfano le condizioni specificate.
- `count(conditions?: FindOptionsWhere<Entity>): Promise<number>`: conta il numero di entità che soddisfano le condizioni specificate.
- `sum(field: string, conditions?: FindOptionsWhere<Entity>): Promise<number>`: calcola la somma dei valori di un campo specificato delle entità che soddisfano le condizioni specificate.
- `createQueryBuilder(alias?: string): SelectQueryBuilder<Entity>`: restituisce un `SelectQueryBuilder` per costruire query SQL in modo programmatico.

Metodi di istanza:

- `save(): Promise<Entity>`: salva l'entità nel database. Se l'entità ha un campo `id` già valorizzato, viene eseguito un `UPDATE`, altrimenti viene eseguito un `INSERT`.
- `remove(): Promise<Entity>`: rimuove l'entità dal database. Viene eseguito un `DELETE`.
- `reload(): Promise<Entity>`: ricarica l'entità dal database. Viene eseguito un `SELECT` in base alla primary key dell'entità.

Un `FindOneOptions<Entity>` è un oggetto che mappa i nomi delle colonne alle condizioni di ricerca, ed è tipizzato tramite il generic `Entity`: Il compilatore Typescript controlla che le colonne sulle quali si effettua la ricerca siano effettivamente presenti nell'entità. Le opzioni che rende disponibili sono:

- `select: FindOptionsSelect<Entity>`: specifica i campi da selezionare. È un oggetto che mappa i campi di un oggetto di tipo `Entity` a `true` o `false`, per selezionare o meno il campo. Se non è specificato, vengono selezionati tutti i campi. Inoltre, non tutti i campi devono essere presenti: `Entity` viene passato attraverso un *partial* Typescript[22].
- `where: FindOptionsWhere<Entity>[]`: specifica le condizioni di ricerca. È un oggetto o un array di oggetti che mappano i campi di un oggetto di tipo `Entity` a delle operazioni di ricerca, e possono anche essere annidati per specificare condizioni complesse, con operatori logici e di confronto. Di default le operazioni sono in OR. Ogni campo di tipo `T` di `Entity` è essere mappato a un oggetto di tipo `T | FindOperator<T>` che specifica l'operatore di confronto.
- `relations: FindOptionsRelations<Entity>`: specifica le relazioni da caricare in j oin insieme all'entità principale. È un oggetto che ha campi parziali di `Entity` che sono stati decorati con `@ManyToOne`, `@OneToOne`, `@OneToMany` o `@ManyToMany`.

[22]Un oggetto di tipo `Partial<T>` è un oggetto che ha le stesse proprietà di `T`, ma con valori opzionali. Nelle definizioni di TypeORM i campi opzionali sono ottenuti con `[P in keyof Entity]?:`, un *mapped type* al quale viene applicato l'operatore `?`.

- `order: FindOptionsOrder<Entity>`: specifica l'ordine con cui le colonne vengono restituite, con degli string literals: `"asc"`, `"desc"`. È analogo a `ORDER BY` in SQL.

La classe `FindManyOptions<Entity>` estende `FindOneOptions<Entity>` e aggiunge le seguenti opzioni:

- `take: number`: specifica il numero massimo di entità da restituire. È analogo a `LIMIT` in SQL.
- `skip: number`: specifica l'offset delle entità da restituire. È analogo a `OFFSET` in SQL. In combinazione con `take` permette di paginare i risultati.

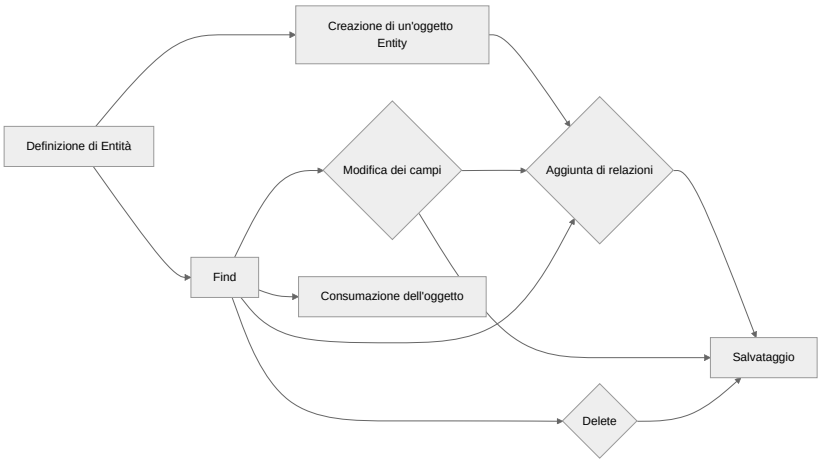
Un `FindOperator<T>` è un oggetto che mappa un operatore di confronto a un valore di tipo `T`, e permette di specificare condizioni logiche:

- `And<T>`: Specifica che tutte le condizioni devono essere soddisfatte.
- `Or<T>`: Specifica che almeno una delle condizioni deve essere soddisfatta.
- `Not<T>`: Specifica che la condizione deve essere negata.

e di confronto:

- `Equal<T>`: specifica che il campo deve essere uguale al valore specificato.
- `LessThan<T>`: specifica che il campo deve essere minore del valore specificato.
- `LessThanOrEqual<T>`: specifica che il campo deve essere minore o uguale al valore specificato.
- `MoreThan<T>`: specifica che il campo deve essere maggiore del valore specificato.
- `MoreThanOrEqual<T>`: specifica che il campo deve essere maggiore o uguale al valore specificato.
- `In<T>`: specifica che il campo deve essere uno dei valori specificati.
- `Like<T>`: specifica che una stringa deve corrispondere ad un pattern specificato. Il pattern può contenere i caratteri jolly `%` e `_`.

Il workflow per operazioni CRUD in Active record segue il diagramma:



Un esempio sulle entità definite nel paragrafo Relazioni molti a molti, è il seguente:

```typescript
// Aggiungi un nuovo utente
const newUser = new User();
newUser.username = "bob";
await newUser.save();

// Trova tutti gli utenti
const allUsers = await User.find();

// Trova l'utente con username "alice"
const alice = await User.findOne({ where: { username: "alice" } });

// Trova un post caricando gli utenti a cui piace ed aggiungi Alice
const postWithLikes = await Post.findOne({
    where: { id: 1 },
    relations: { likedBy: true },
});
postWithLikes.likedBy.push(alice);
await postWithLikes.save();

// Carica gli utenti che hanno messo "mi piace" ai post di Alice o di Bob
const authors = await User.find({
    where: [{ username: In("alice", "bob") }],
    relations: { posts: { likedBy: true } },
});

const usersWhoLikedAuthorsPostsWithDuplicates: User[] = authors.flatMap((author) =>
    author.posts.flatMap((post) => post.likedBy)
);

const usersWhoLikedAuthorsPosts = [...new Set(usersWhoLikedAuthorsPostsWithDuplicates)]
```

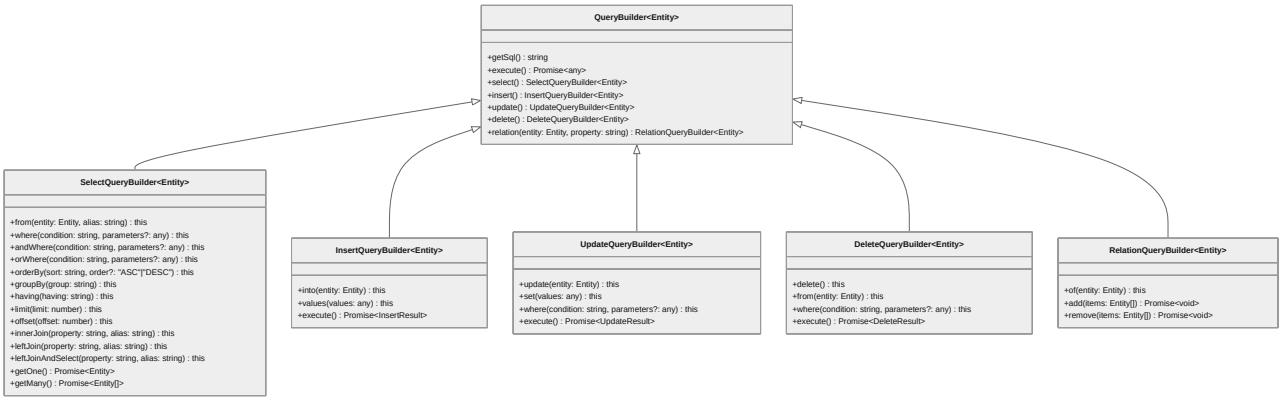
Query builder

Per efficientare una query complessa, che fa join su più tabelle, si può usare l'API Query Builder, che fa uso del pattern implementativo *Builder* per costruire una query SQL vincolandone la grammatica ai metodi disponibili.

Si parte dall'oggetto DataSource e si ottiene una repository relativa all'entità, con `getRepository(Entity)`, da questa si accede a `createQueryBuilder(), che restituisce un'istanza di`QueryBuilder‘, i cui metodi principali sono:

- `execute(): Promise<any>`: esegue la query e restituisce il risultato.
- `select(): SelectQueryBuilder<Entity>`: costruisce una query di selezione.
- `insert(): InsertQueryBuilder<Entity>`: costruisce una query di inserimento.
- `update(): UpdateQueryBuilder<Entity>`: costruisce una query di aggiornamento.
- `delete(): DeleteQueryBuilder<Entity>`: costruisce una query di eliminazione.
- `relation(entity: Entity, property: string): RelationQueryBuilder<Entity>`: costruisce una query per manipolare relazioni.

Ognuno di questi builder estende `QueryBuilder<Entity>` ed ha a disposizione dei metodi che ricalcano la sintassi SQL.



Per query complesse, Query Builder si basa su *alias* per le tabelle coinvolte, cioè stringhe che identificano le tabelle in modo univoco all'interno della query. Gli alias sono passati come argomento ai metodi di join e di selezione, e vengono usati per specificare le colonne e le condizioni di ricerca. Questo meccanismo permette di costruire query SQL con più livelli di join, ma è prono ad errori di battitura, potenzialmente difficili da individuare durante un debug. L'utilizzo di costanti per gli alias, definite all'inizio del file, può aiutare a ridurre il rischio di errori.

Lo stesso esempio del paragrafo Active record, dove le entità estendono `BaseEntity` per rendere disponibile il metodo `createQueryBuilder()`, può essere riscritto con Query Builder come segue:

```
const USER = "user";
const POST = "post";
// Aggiungi un nuovo utente
await User.createQueryBuilder()
    .insert()
    .into(User)
    .values({ username: "bob" })
    .execute();

// Trova tutti gli utenti
const allUsers = await User.createQueryBuilder().getMany();

// Trova l'utente con username "alice"
const alice = await User.createQueryBuilder(USER)
    .select()
    .where(USER + ".username = :username", { username: "alice" })
    .getOne();

// Trova un post caricando gli utenti a cui piace ed aggiunge Alice
const postWithLikes = await Post.createQueryBuilder(POST)
    .leftJoinAndSelect(POST + ".likedBy", "likedBy")
    .where(POST + ".id = :id", { id: 1 })
```

```
    .getOne();

await Post.createQueryBuilder()
    .relation(Post, "likedBy")
    .of(postWithLikes)
    .add(alice);

// Carica gli utenti che hanno messo "mi piace" ai post di Alice o di Bob
const usersWhoLikedAuthorsPosts = await User.createQueryBuilder("user")
    .innerJoin("user.likedPosts", "likedPost")
    .innerJoin("likedPost.author", "author")
    .where("author.username IN (:...usernames)", {
        usernames: ["alice", "bob"],
    })
    .distinct(true)
    .getMany();
```

Le differenze principali con Active Record sono:

- Query Builder è più verboso, ma permette di costruire query complesse in modo più flessibile, che sono più efficienti e vicine al modello relazionale.
- L'aggiunta di entità a relazioni molti a molti in Active Record avviene in memoria, e non persiste fino a quando non si chiama `save()` sull'entità principale. Query Builder manipola direttamente la relazione nel database.
- In Query Builder si possono applicare metodi di aggregazione, come `distinct()`, invece che scorrere in memoria un array di risultati intermedi, che contiene anche elementi da scartare.

Query SQL raw

Infine TypeORM rende disponibile un'API per eseguire query SQL interpolate direttamente in stringhe con il metodo `query(query: string, parameters?: any[])` di `EntityManager`. L'unica astrazione fornita in questo caso è la sanificazione dei parametri, che previene attacchi di SQL injection. Lo stesso esempio riportato sopra può essere riscritto con query SQL come segue:

```
// Aggiungi un nuovo utente
await dataSource.query(`
  INSERT INTO "user" ("username")
  VALUES ('bob');
`);

// Trova tutti gli utenti
const allUsers = await dataSource.query(`
  SELECT * FROM "user";
`);

// Trova l'utente con username "alice"
const [alice] = await dataSource.query(`
  SELECT * FROM "user"
```

```
  WHERE "username" = 'alice'
  LIMIT 1;
`);

// Trova un post caricando gli utenti a cui piace
const [postWithLikes] = await dataSource.query(`
  SELECT p.*, u.*
  FROM "post" p
  LEFT JOIN "post_likes" pl ON p.id = pl.post_id
  LEFT JOIN "user" u ON pl.user_id = u.id
  WHERE p.id = 1;
`);

// Aggiungi Alice ai like del post
await dataSource.query(
    `
  INSERT INTO "post_likes" ("post_id", "user_id")
  VALUES (?, ?);`,
    [postWithLikes.id, alice.id]
);

// Carica gli utenti che hanno messo "mi piace" ai post di Alice o di Bob
const usersWhoLikedAuthorsPosts = await dataSource.query(`
  SELECT DISTINCT u.*
  FROM "user" u
  INNER JOIN "post_likes" pl ON u.id = pl.user_id
  INNER JOIN "post" p ON pl.post_id = p.id
  INNER JOIN "user" author ON p.author_id = author.id
  WHERE author.username IN ('alice', 'bob');
`);
```

Query come queste massimizzano le prestazioni che si possono ottenere con un database relazionale, al costo di una maggiore probabilità di errori di battitura. L'utilizzo di costanti per le tabelle e le colonne, definite all'inizio del file, può aiutare a ridurre questo rischio.

Capitolo 3

Esempi di Applicazioni Web basate su Nuxt e TypeORM

Vengono esposte delle soluzioni progettuali ed implementative per la realizzazione di un'applicazione Web con le tecnologie dettagliate nel capitolo precedente, Nuxt e TypeORM, in combinazione con i servizi cloud AWS, che ho approfondito durante il tirocinio curriculare presso l'azienda Soluzioni Futura s.r.l., ora Polarity s.r.l.

Il progetto realizzato per questo lavoro di tesi include:

- Progettazione delle infrastrutture cloud mediante codice *Cloudformation*, secondo il modello a container e serverless, e di script di integrazione continua con *Github Actions*.
- Progettazione, per entrambe le infrastrutture, di sistemi di integrazione di TypeORM con Nuxt.
- Deploy e test di performance di applicazioni di esempio con Nuxt e TypeORM, su entrambe le infrastrutture.

3.1 Architettura del cloud e integrazione continua

Amazon Web Services è una piattaforma che offre una *PaaS*, platform as a service, dove sono messi a disposizione servizi di calcolo, di storage e di database; ma anche una *IaaS*, infrastructure as a service, che permette di configurare reti di calcolatori virtuali accessibili via internet.

Per iniziare ad utilizzare AWS è necessario registrarsi (come root user dell'account) accedendo alla dashboard online (https://aws.amazon.com/it/console/).

3.1.1 Progettazione dell'infrastruttura dei servizi cloud AWS

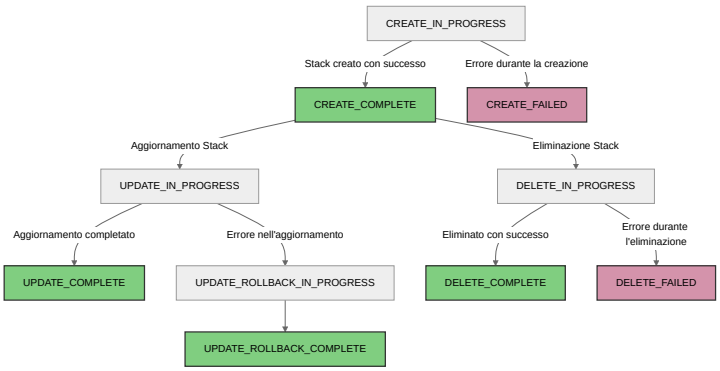
Parte del progetto è stata l'avviamento dell'infrastruttura AWS per mezzo di codice di marcatura `yaml` con Cloudformation, un servizio che permette di gestire risorse in AWS in modo dichiarativo. Cloudformation permette di creare interi stack nei quali si possono creare e collegare servizi AWS.

Il seguente è un esempio concettuale di template Cloudformation `yaml`, dove vengono presi in input username e password, poi si avviano due risorse: un server linux virtuale EC2 e un'i-

stanza di database RDS che viene configurata con i parametri in ingresso tramite `!Ref`. L'istanza EC2 verrà creata solo dopo che l'istanza RDS sarà stata avviata, per via della direttiva `!GetAtt "RDSInstance.Endpoint.Address"`, così il suo accesso al database sarà garantito. Infine viene restituito l'indirizzo IP pubblico dell'istanza EC2.

```yaml
Description: "Deploy a database with provided username and password"
Parameters:
    DBUsername:
        Type: String
        Description: Database master username
    DBPassword:
        Type: String
        Description: Database master password
        NoEcho: true

Resources:
    RDSInstance:
        Type: AWS::RDS::DBInstance
        Properties:
            MasterUsername: !Ref "DBUsername"
            MasterUserPassword: !Ref "DBPassword"

    EC2Instance:
        Type: AWS::EC2::Instance
        Properties:
            Environment:
                Variables:
                    DB_HOSTNAME: !GetAtt "RDSInstance.Endpoint.Address"
                    DB_USERNAME: !Ref "DBUsername"
                    DB_PASSWORD: !Ref "DBPassword"

Outputs:
    EC2Address:
        Description: "Public IP address of the EC2 instance"
        Value: !GetAtt "EC2Instance.PublicIp"
```

Uno stack Cloudformation si può trovare in vari stati, mostrati nel diagramma:



Di seguito sono esposte due architetture diverse ed essenziali, in quanto forniscono una base di partenza funzionante e che può essere personalizzata per soddisfare le esigenze di una qualsiasi applicazione e del suo team di sviluppo modificando i rispettivi template Cloudformation.

Le due architetture sono ospitate dalla *Vpc* (Virtual Private Cloud, un modo per isolare le risorse dei vari utenti di AWS) di default, che dispone di tre subnet pubbliche collegate ad un Internet Gateway. Entrambe le architetture espongono servizi HTTP, ma non forniscono un DNS personalizzato, che si potrebbe aggiungere con il servizio Route 53 di AWS. In più non sono stati configurati dei sistemi di accesso sicuro a database: l'unica modalità di accesso prevista è quella tramite username e password. Per effettuare migrazioni si è lasciato il database accessibile tramite subnet pubblica, ma lo scenario di produzione richiederebbe di configurare una subnet privata alla quale gli sviluppatori dovrebbero accedere tramite un bridge SSH che passa per un server *bastion* localizzato nella stessa subnet e che espone quindi servizi di accesso remoto.

Architettura basata su containers

L'architettura basata su container AWS Elastic Cloud Service (ECS) è una soluzione che permette di gestire container Docker che eseguono una certa applicazione in un cluster di macchine virtuali EC2. Questa architettura è simile a quella di un'applicazione monolitica:

- ECS richiede minore configurazione di server singoli. I container sono immagini autocontenute che possono essere avviate senza preparare l'ambiente a mano o mediante script. Si può aggiornare l'immagine del container senza dover riavviare l'intera macchina virtuale.
- La scalabilità è più semplice, in quanto si può configurare il numero di container in base al carico di lavoro. Ogni container è associato ad una task, e queste si comportano in maniera idempotente. È facile impostare un bilanciamento del carico tra le task.
- I prezzi sono fissi in base alle risorse che si utilizzano. Se si sceglie di avere del bilanciamento aggiuntivo mediante *autoscaling* si pagherà per le risorse utilizzate, ma bisogna considerare dei costi di base per tenere in esecuzione almeno un container.
- Nonostante l'autoscaling, il modello di server è di tipo **stateful**, in quanto i container rimangono attivi per un tempo indefinito.

Sono stati scritti due stack, per disaccoppiare l'infrastruttura di base e permanente da quella che può essere personalizzata.

`infrastructure.yml`, riguarda l'infrastruttura di base, ed include:

- Una repository ECR per ospitare le immagini Docker, che verranno costruite e caricate da Github Actions.
- Un cluster ECS con un servizio che esegue i container.
- Un load balancer per esporre il servizio, composto da più task, in maniera uniforme.
- Un Security Group per permettere il traffico HTTP da e verso il load balancer.

`service.yml` riguarda il servizio, ed include:

- Task definition per il container, quindi la capacità di calcolo e di memoria di ciascun container.

- Un servizio ECS che esegue i container, con un bilanciamento del carico tra le task, e che assegna a ciascuno 512MB di memoria e 0.25 vCPU.
- Un servizio di log Cloudwatch per monitorare i log dei container in maniera unificata.
- Un servizio di database RDS, con le credenziali di accesso passate come variabili d'ambiente. La connessione delle task al database avviene tramite variabili d'ambiente come mostrato prima.

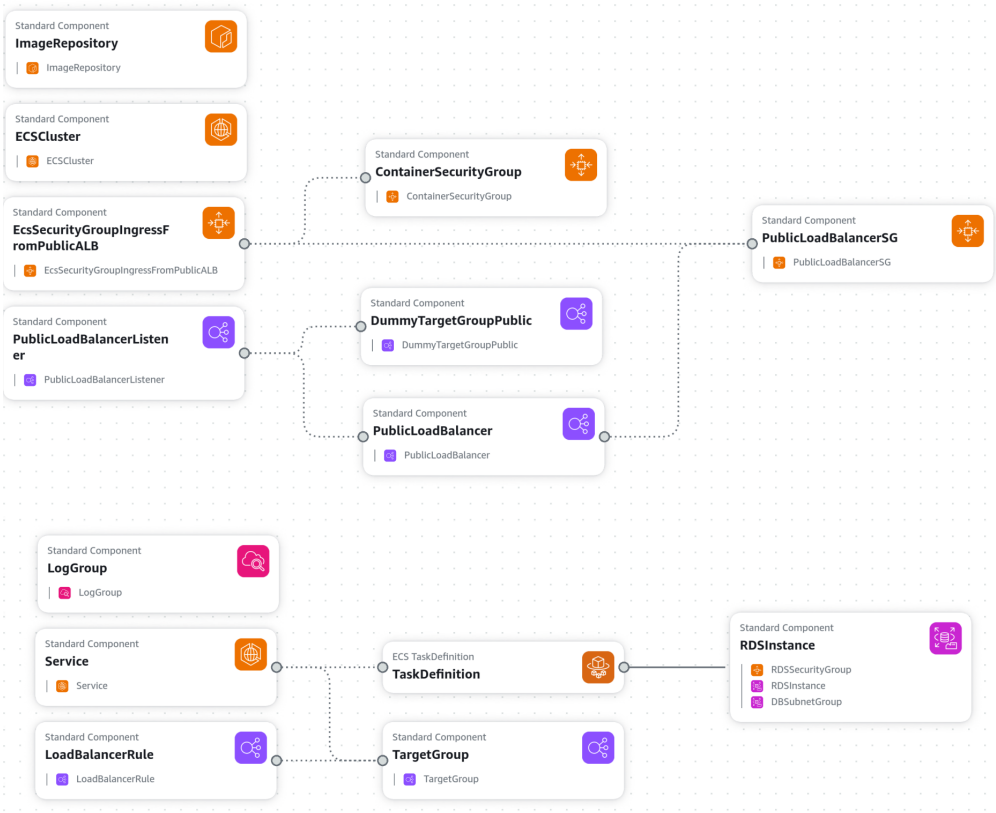


Diagramma dell'architettura basata su containers: sopra lo stack `infrastructure.yml`, sotto `service.yml`

La repository con il template relativo all'architettura basata su containers realizzato per il progetto di tesi è ospitata su Github all'indirizzo github.com/iacobucci/cfn-nuxt-typeorm-ecs-rds.

Architettura serverless

La seconda architettura proposta, basata su funzioni AWS Lambda, è una soluzione serverless. Con serverless si intende[1] l'esecuzione di codice in container di calcolo senza stato, avviati tramite eventi, effimeri (potrebbero essere eliminati dopo una sola invocazione), e completamente gestiti dal provider cloud. Questo modello è adatto per applicazioni che richiedono scalabilità automatica

[1]Citando la definizione "2" in Serverless architectures - articolo di Mike Roberts sul blog di Martin Fowler.

e che non necessitano di server attivi per lunghi periodi di tempo. Le funzioni Lambda hanno le seguenti caratteristiche:

- La configurazione di una Lambda è minima: basta caricare un archivio `zip` del codice da eseguire e specificare un runtime tra i supportati (Node.js, Python, Java, ecc.).
- La scalabilità è la migliore possibile: il provider cloud si occupa di avviare nuove istanze di Lambda in base al carico di lavoro.
- I costi sono basati sul tempo di esecuzione e sulle risorse utilizzate. Se il codice non viene eseguito, non si pagherà nulla.
- Il modello di server è di tipo **stateless**, in quanto le funzioni Lambda non mantengono lo stato tra le invocazioni. Questo significa che non si può mantenere una connessione attiva al database.
- Il limite di esecuzione di una Lambda è di 15 minuti. Se il codice richiede più tempo, si dovrà spezzare la funzione in più Lambda.
- Soffrono del problema del *cold start*: la prima invocazione di una funzione Lambda può richiedere più tempo rispetto alle successive, in quanto il provider cloud deve avviare un container di calcolo e caricare il codice della funzione. Le invocazioni successive saranno più veloci, in quanto il container sarà riutilizzato, seppure per breve tempo. AWS offre un'opzione chiamata Provisioned Concurrency, che tiene istanze della tua Lambda pronte all'uso, mitigando il problema del cold start: con questa opzione si possono mantenere funzioni in un certo numero come sempre attive, ma bisogna considerare che questo comporta un costo fisso.

È stato configurato lo stack `serverless.yml`, che include:

- Una funzione lambda che esegue il codice dell'applicazione. La funzione ha a disposizione 256MB di memoria.
- Un security group per permettere il traffico HTTP da e verso la funzione.
- Un database AWS Aurora, con le credenziali di accesso passate come variabili d'ambiente. La connessione della funzione al database avviene tramite variabili d'ambiente come mostrato prima. Aurora ha una scalabilità automatica pensata per sistemi serverless e un'alta disponibilità. Questo aiuta a ridurre il costo di gestione del database.
- Un proxy RDS interposto tra la funzione e il database, per evitare di sovraccaricare il database con troppe connessioni aperte. Il proxy è configurato per gestire un pool di connessioni al database e riutilizzarle.

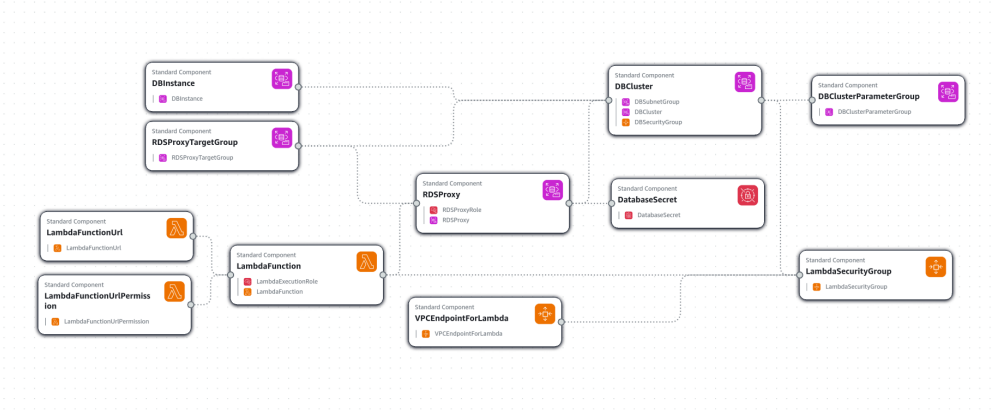


Diagramma dell'architettura basata su funzioni serverless: lo stack `serverless.yml`

La repository con il template relativo all'architettura basata su funzioni serverless realizzato per il progetto di tesi è ospitata su Github all'indirizzo github.com/iacobucci/cfn-nuxt-typeorm-lambda-aurora.

3.1.2 Continuous Integration e Continuous Deployment con Github Actions

Per iniziare a pubblicare la loro applicazione, il team di sviluppo potrà clonare nell'account della propria organizzazione uno dei due *template Github* linkati sopra, poi iniziare a configurare l'account AWS per la connessione alla repository. Per non passare dalla dashboard di AWS, è suggerito un procedimento che richiede:

1. L'installazione della CLI di AWS da un ambiente di linea di comando POSIX-compatibile con Python 3.6 o superiore installato:

```
pip3 install --user awscli
```

Ed il login dalla CLI al proprio account AWS con la procedura guidata:

```
aws configure
```

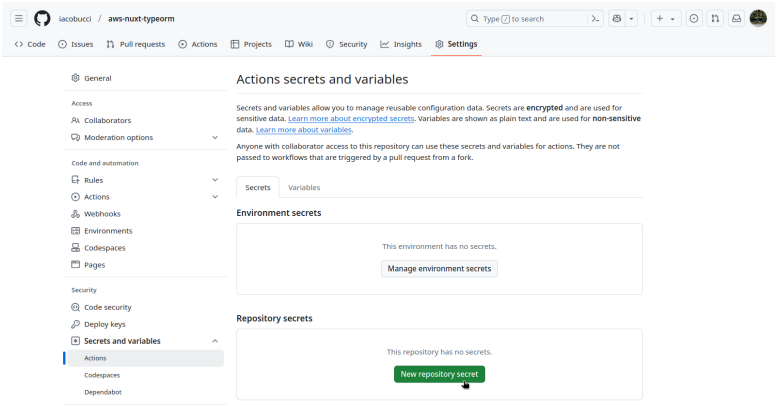
2. L'avvio di un deploy Cloudformation, da eseguire una sola volta, che creerà un ruolo *IAM* con permessi limitati alle operazioni normalmente eseguite dall'integrazione continua ed anche un'identità federata OpenID Connect per limitare quelle operazioni alla sola repository Github in questione.

```
export GIHTUB_ORG=iacobucci
export REPOSITORY_NAME=aws-nuxt-typeorm
aws cloudformation deploy \
    --stack-name github-actions-cloudformation-deploy-setup \
    --template-file cloudformation/setup.yml \
    --capabilities CAPABILITY_NAMED_IAM \
    --region eu-central-1 \
    --parameter-overrides GitHubOrg=$GITHUB_ORG RepositoryName=$REPOSITORY_NAME
```

3. Configurare la propria repository per l'integrazione continua, aggiungendo:

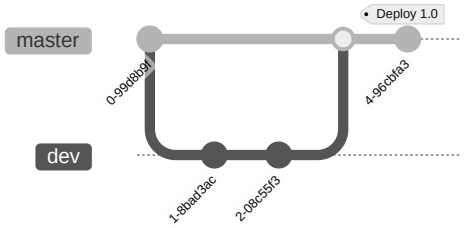
- `AWS_ACCOUNT_ID`: l'ID dell'account AWS che intendono utilizzare.
- `DB_NAME`: il nome principale del database.
- `DB_PORT`: la porta TCP usata durante le comunicazioni con il database.
- `DB_USERNAME`: l'username principale del database.
- `DB_PASSWORD`: la password dell'username principale.

nei *Secrets* di Github, come mostrato in figura. I Secrets sono variabili che non vengono esposte nel codice sorgente, ed una volta aggiunte non possono essere visualizzate nuovamente.



Impostazione dei secrets in una repository GitHub

4. Iniziare a scrivere codice su branch di sviluppo.



Ad un push su `master` inizierà il workflow di Github Actions, che si occuperà di fare:

- Nel caso dell'architettura basata su container:

1. Checkout del codice sorgente.
2. Login ad AWS.
3. Deploy dello stack `infrastructure.yml` con Cloudformation.
4. Login al registry ECR creato dallo stack.
5. Build, tag e push dell'immagine Docker su ECR. Qui saranno disponibili le varie versioni di produzione dell'applicazione.
6. Deploy dello stack `service.yml` con Cloudformation.

7. Stampa dell'url del servizio e del database.

- Nel caso dell'architettura serverless:

1. Checkout del codice sorgente.
2. Login ad AWS.
3. Creazione, se non esiste, di un *bucket* S3 con versionamento per il salvataggio del codice della Lambda. Saranno disponibili anche qui le varie versioni di produzione dell'applicazione.
4. Installazione delle dipendenze.
5. Build del progetto.
6. Creazione di un file zip con il `.output` della build.
7. Caricamento del file zip su S3.
8. Deploy dello stack Cloudformation.
9. Stampa dell'url del servizio e del database.

In questo modo il team di sviluppo potrà concentrarsi sullo sviluppo del codice, mentre l'integrazione continua si occuperà di fare deploy dell'applicazione in produzione. Sarà leggibile anche dalla repository se il deploy è andato a buon fine, e in caso contrario sarà possibile vedere i log dell'errore.

Una prima metrica di performance di queste soluzioni architetturali è il tempo di completamento del workflow che le implementa. Questo può essere monitorato nella dashboard "Actions" della repository.

I dati che ho rilevato, per il progetto di esempio completo e di Nuxt e TypeORM, sono i seguenti:

Architettura	Tempo di creazione	Tempo di aggiornamento
Container	14m 58s	6m 35s
Serverless	14m 41s	1m 20s

3.2 Un'applicazione di esempio con Nuxt e TypeORM

È stata realizzata una semplice applicazione di esempio, che fa uso di query TypeORM con i vari pattern descritti, con il modello di dominio dell'esempio relazioni molti a molti. È stata impostata la modalità di `server-side rendering` in entrambi i casi di deploy: uno sullo stack ECS e l'altro sullo stack Lambda. Si è usata questa applicazione di social networking con:

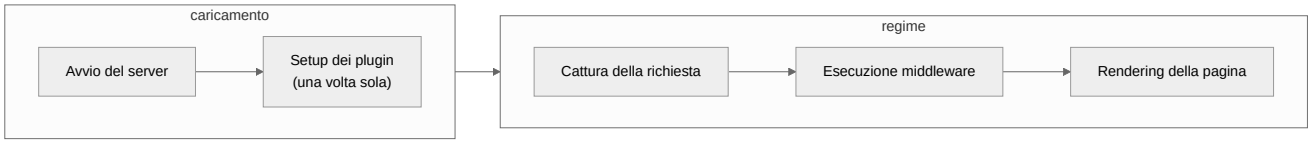
- `/users/[page]`: pagina principale con lista di utenti, che può essere scorsa con degli appositi pulsanti.
- `/user/[username]`: pagina di dettaglio di un utente, con lista dei post scritti.
- `/post/[id]`: pagina di dettaglio di un post, con il contenuto e lista degli utenti che hanno messo "mi piace".

Infine, per testare query più complesse, è stata aggiunta una pagina:

- `/users/whoLikedPostsByAuthors`: pagina che mostra gli utenti che hanno messo "mi piace" ai post di una lista di autori. Query di questo tipo potrebbero essere utili per un eventuale sistema di raccomandazione di post.

3.2.1 Implementazione di TypeORM in Nuxt

Per implementare TypeORM è utile osservare il ciclo di vita di una richiesta HTTP in Nuxt:



Per il motivo che le task di ECS si comportano come server stateful, in quanto dopo l'avvio tendono a rimanere attive fino alla loro terminazione manuale, si può impostare il collegamento al database in un *plugin* Nuxt. Anche in caso di guasti, il servizio ECS si riavvia automaticamente e la continuità del servizio è molto probabile in quanto rimarranno attive altre task, e quella guasta verrà sostituita.

L'istanza di DataSource è esportata da un file Typescript in `~/server/utils` per essere utilizzata in altri moduli `~/server/` del progetto. È come una funzione asincrona `initialize()`

```
export const AppDataSource = new DataSource(options);

export async function initialize() {
    try {
        if (!AppDataSource.isInitialized) {
            await AppDataSource.initialize();
            console.log("Typeorm inizializzato", {
                type: AppDataSource.options.type,
                database: AppDataSource.options.database,
            });
        }
    } catch (error) {
        console.error("Errore inizializzazione Typeorm", error);
        throw error;
    }
}
```

Il plugin in `~/server/plugins/typeorm.ts` ne fa uso:

```
import { AppDataSource, initialize } from "~/server/utils/datasource";

export default defineNitroPlugin(async () => {
    initialize();
});
```

Per servizi stateless invece l'approccio è diverso: ad ogni richiesta bisogna assicurarsi che la connessione al database sia attiva. Non basterebbe inizializzare la connessione all'avvio della funzione Lambda, per via delle limitazioni di tempo di esecuzione. Ad ogni richiesta, bisogna verificare che la connessione sia attiva, e in caso contrario inizializzarla. Una funzione come la seguente può essere utilizzata in un middleware lato server Nuxt. È una funzione ricorsiva che implementa un semplice algoritmo di *backoff esponenziale* in caso di fallimento.

```
export async function ensureDataSource(retryCount = 3, delayMs = 1000) {
    try {
        if (!AppDataSource.isInitialized) {
            await AppDataSource.initialize();
            console.log("TypeORM inizializzato");
        } else {
            await AppDataSource.query("SELECT 1"); // semplice query di test
        }
    } catch (error) {
        console.error(
            `Errore durante la verifica o inizializzazione di TypeORM (tentativo ${
                4 - retryCount
            }/3)`,
            error
        );

        if (retryCount > 0) {
            if (AppDataSource.isInitialized) {
                await AppDataSource.destroy();
            }
            await new Promise((resolve) => setTimeout(resolve, delayMs)); // Aspetta
             ↪  prima di riprovare
            return ensureDataSource(retryCount - 1, delayMs * 2); // Backoff esponenziale
        } else {
            console.error(
                "Esauriti i tentativi di inizializzazione di TypeORM"
            );
            throw error;
        }
    }
}
```

Le opzioni di connessione sono state in entrambe i casi configurate come segue, per ottenere il collegamento descritto nella sezione di progettazione:

```
const options =
    process.env.NODE_ENV === "production"
        ? {
                type: "postgres",
                host: process.env.DB_HOSTNAME,
                database: process.env.DB_NAME,
                port: parseInt(process.env.DB_PORT || "5432"),
                username: process.env.DB_USERNAME,
                password: process.env.DB_PASSWORD,
                ssl: true,
                synchronize: false,
                logging: true,
                entities, // array di classi delle entità da collegare
          }
        : // configurazione di sviluppo e di test...
```

Ma per il caso serverless, con utilizzo di pool di connessioni sono state selezionate delle opzioni aggiuntive:

```
{
    extra: {
        max: 1000,
        min: 10,
        connectionTimeoutMillis: 30000,
        keepAlive: true, // con questo si evita la riconnessione ad ogni richiesta
        keepAliveInitialDelayMillis: 5000,
        query_timeout: 10000,
    },
}
```

Inoltre, seppure nel caso basato su container sia sconsigliato, per non incombere in problemi di disallineamento delle strutture dati, è comunque possibile impostare `synchronize: true` per sincronizzare automaticamente lo schema del database con le entità definite in TypeORM. Nel caso serverless `synchronize` deve rimanere `false`, in quanto effettuare operazioni di migrazione ad ogni richiesta che risveglia una nuova Lambda renderebbe inutilmente lenta l'esecuzione del codice, oltre a portare a problemi di concorrenza.

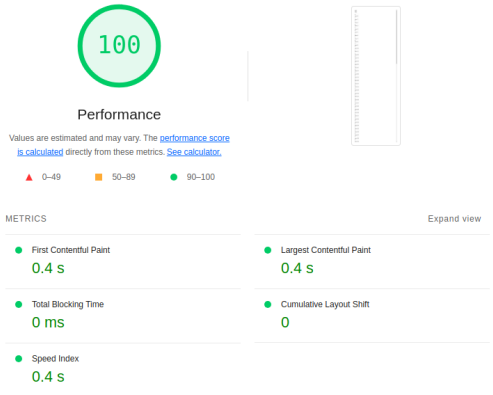
3.3 Analisi di performance e sicurezza

Per valutare la riuscita dell'implementazione delle tecnologie scelte, sono stati effettuati dei test sull'efficacia del rendering server side, per confermare che questa tecnica è in grado di mitigare i problemi descritti nel capitolo 1. Poi sono stati effettuati dei test quelle query più complesse, per valutare l'efficacia di TypeORM.

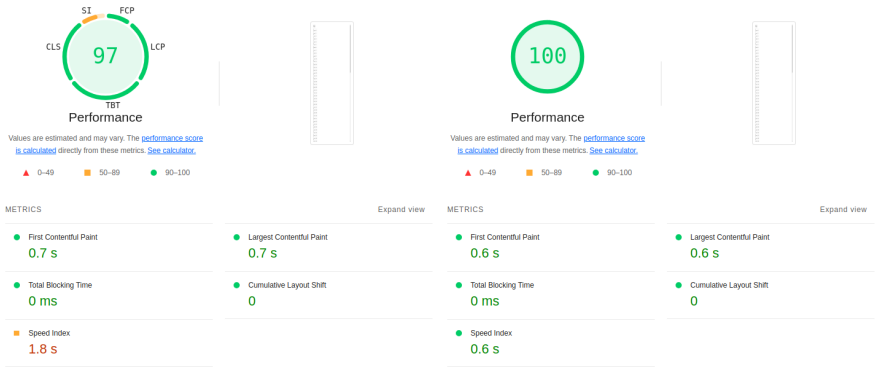
3.3.1 Audit di rendimento lato client

È stato usato lo strumento di profilazione Lighthouse di Chromium, per fare un audit della principale. Questa fa due redirect lato server: uno da `/` a `/users` fino a `/users/1`, per mostrare la prima pagina di tutti gli utenti.

Il risultato, nel caso di ECS, è stato:



E nel caso di Lambda:



Performance del SSR per l'architettura Lambda. A sinistra in caso di *cold start*, a destra in caso di *warm start*.

L'architettura ECS ha, in media, prestazioni migliori in termini di tempo, complice il fatto che c'è un numero minimo di task che rimangono sempre attive ed in ascolto di richieste.

Con questo test si è anche mostrato come il first contentful paint e il largest contentful paint coincidano, indicando che la pagina è pronta per l'utente senza bisogno di ulteriori caricamenti. Assieme all'ottimizzazione della SEO, il problema principale del capitolo 1 è stato risolto.

3.3.2 Test di stress per Active record e Query Builder

Per fare un test delle prestazioni dell'applicazione di esempio sotto elevato carico, sono stati inseriti dei dati *mock*, che includono:

- 10000 utenti.
- 100000 post.
- 1000000 "mi piace" ai post.

Per ogni configurazione sono stati effettuati dei test di stress con lo strumento `hey`[2], che ha eseguito 200 richieste HTTP con 50 thread contemporanei agli endpoint:

- `/users/whoLikedPostsByAuthorsQueryBuilder`, che esegue la query numero 5 descritta nel paragrafo Query Builder
- `/users/whoLikedPostsByAuthorsActiveRecord`, che esegue la query numero 5 descritta nel paragrafo Active Record

con parametri: `?authors=user-1,...,user-10`, quindi scegliendo 10 autori di post.

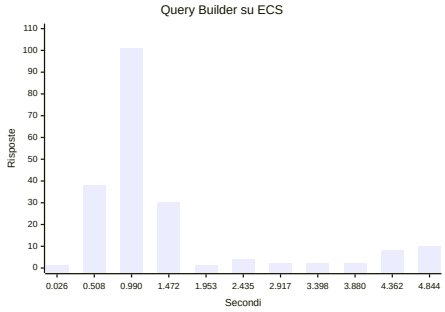
La risposta HTTP ad entrambe le query consiste in una pagina che contiene 927 `RowsUser`, cioè dei componenti Vue che indicano il nome utente e contengono un Nuxt Link a `/user/[username]`

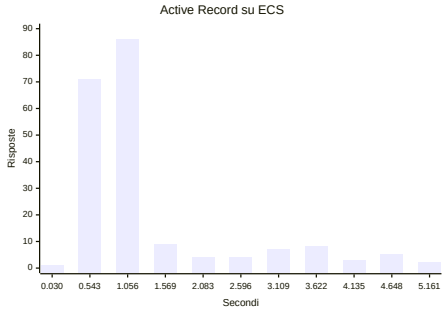
I risultati, in termini di quante risposte sono state ricevute dopo un certo tempo, sono stati raccolti in grafici a barre con intervalli temporali omogenei per ogni test. Le risposte sono state

[2] github.com/rakyll/hey - il repository di hey su Github.

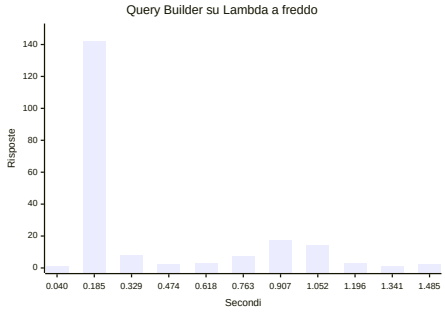
sempre conformi alle attese, e non sono state riscontrate anomalie, riscontrando un 100% di status code “200”.

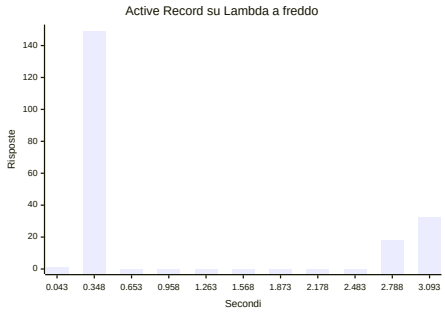
I risultati, per l’architettura basata su container, sono stati:



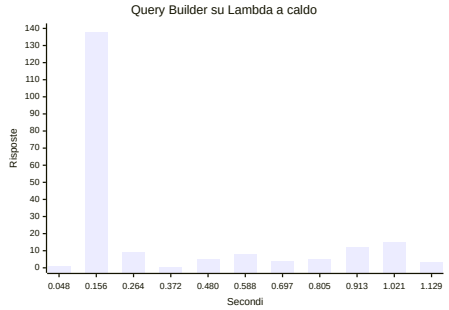


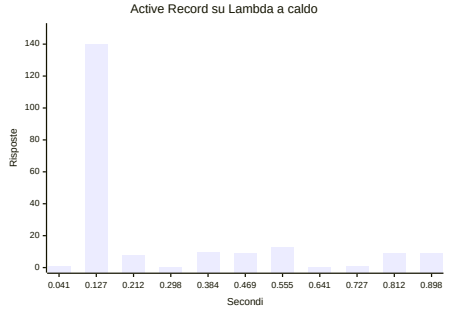
Mentre per l’architettura serverless, con funzioni Lambda “a freddo”, che quindi devono rendere conto al tempo di avvio della funzione:





E per Lambda “a caldo”, cioè con funzioni già avviate:





Esaminando il codice SQL effettivamente eseguito sul database, per Query Builder si ha:

```sql
SELECT DISTINCT "user"."id" AS "user_id",
                "user"."username" AS "user_username"
FROM "user" "user"
INNER JOIN "post_liked_by_user" "likedPost_user" ON "likedPost_user"."userId"="user"."id"
INNER JOIN "post" "likedPost" ON "likedPost"."id"="likedPost_user"."postId"
INNER JOIN "user" "author" ON "author"."id"="likedPost"."authorId"
WHERE "author"."username" IN ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10)
```

e per Active Record:

```sql
SELECT "User"."id" AS "User_id",
       "User"."username" AS "User_username",
       "User__User_posts"."id" AS "User__User_posts_id",
       "User__User_posts"."content" AS "User__User_posts_content",
       "User__User_posts"."authorId" AS "User__User_posts_authorId",
       "User__User_posts__User__User_posts_likedBy"."id" AS
        ↪  "User__User_posts__User__User_posts_likedBy_id",
       "User__User_posts__User__User_posts_likedBy"."username" AS
        ↪  "User__User_posts__User__User_posts_likedBy_username"
FROM "user" "User"
LEFT JOIN "post" "User__User_posts" ON "User__User_posts"."authorId"="User"."id"
LEFT JOIN "post_liked_by_user"
 ↪  "User__User_posts_User__User_posts__User__User_posts_likedBy" ON
 ↪  "User__User_posts_User__User_posts__User__User_posts_likedBy"."postId"="User__User_posts"."id"
```

```
11 LEFT JOIN "user" "User__User_posts__User__User_posts_likedBy" ON
   ↪  "User__User_posts__User__User_posts_likedBy".
12     "id"="User__User_posts_User__User_posts__User__User_posts_likedBy"."userId"
13 WHERE ("User"."username" IN ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10))
```

Si evince che in entrambi i casi le query al database fanno 3 `JOIN`. La versione generata dalla `.find()` fa uso di più *alias*, ma non è necessariamente più lenta ad eseguire sul database.

Si può notare come l'architettura ECS proposta abbia tempi di risposta più variabili rispetto a quella serverless, che in generale ha prestazioni migliori: ECS ha risposto al 75% delle richieste in meno di un secondo, mentre Lambda ha risposto in meno di un secondo nel 95% dei casi in tutte le configurazioni tranne in quella "Active Record a freddo", che ha avuto il 50% di risposte in meno di un secondo.

Inoltre, a parità di configurazione le velocità di query Query Builder e Active Record sono molto vicine, e quest'ultima in certi casi è anche più veloce.

I risultati di performance ottenuti si spiegano per via dell'uso efficiente del pattern Active Record con le API Repository. Entrambe le query fanno una sola `await` per ottenere i risultati. Inoltre, per la presenza del pool di connessioni fornita da RDS Proxy, nel caso di Lambda non si è verificato il problema di sovraccarico del database con troppe connessioni aperte.

3.3.3 Test di sicurezza e vulnerabilità

Per gli stessi endpoint testati in termini di performance, sono stati effettuati test con `sqlmap`[3], uno strumento di test di sicurezza automatizzati per database SQL. Con i test effettuati, tra cui:

- "AND boolean-based blind - WHERE or HAVING clause", che manipola una condizione booleana in una clausola `WHERE` o `HAVING` per inferire informazioni sul database in modo cieco.
- "UNION SELECT", che tenta di recuperare informazioni da altre tabelle.
- "Time-based blind", che tenta di inferire informazioni sul database in base al tempo di risposta.
- "Stacked queries", che tenta di eseguire più query in una sola richiesta.
- "Error-based - WHERE or HAVING clause", che tenta di ottenere informazioni sul database in base agli errori generati.

Per entrambe le API TypeORM, non è stata rilevata alcuna vulnerabilità.

[3] sqlmap.org - il sito ufficiale di sqlmap.

Conclusioni

L'obiettivo del lavoro di tesi è stato quello di presentare delle soluzioni implementative ed architetturali per Applicazioni Web, declinabili in progetti di vario genere, da semplici blog statici a complessi sistemi di gestione dati. L'efficacia di queste soluzioni, che sono state proposte come continuazioni naturali delle linee evolutive dello sviluppo Web, è stata argomentata dopo aver effettuato test di performance e di sicurezza su applicazioni di esempio.

Le tecnologie descritte sono il framework Nuxt, che permette di creare applicazioni orientate a componenti Vue.js anche implementando Server Side Rendering, poi la libreria TypeORM, che fornisce astrazioni per la gestione di database relazionali in TypeScript. É stata dettagliata anche l'infrastruttura cloud AWS, sulla quale ho lavorato durante il Tirocinio Curriculare, che permette di scalare le risorse in base alle necessità.

Dalle analisi effettuate si possono trarre le seguenti conclusioni:

Il server side rendering di **Nuxt** è efficace per ottenere ottimi risultati di Largest Contentful Paint, Search Engine Optimization, oltre che per impostare un progetto ben strutturato ed estendibile. Con il supporto nativo a Typescript e la possibilità di creare componenti riutilizzabili, Nuxt fornisce un ambiente di sviluppo completo di convenzioni utili a coordinare lo sviluppo di codice collaborativo.

Nonostante che React sia la libreria di componenti reattivi Javascript più diffusa e documentata, Vue.js rimane un'alternativa molto apprezzata per la sua espressività e per la sua curva di apprendimento più dolce, e Nuxt è il framework di punta per questa tecnologia.

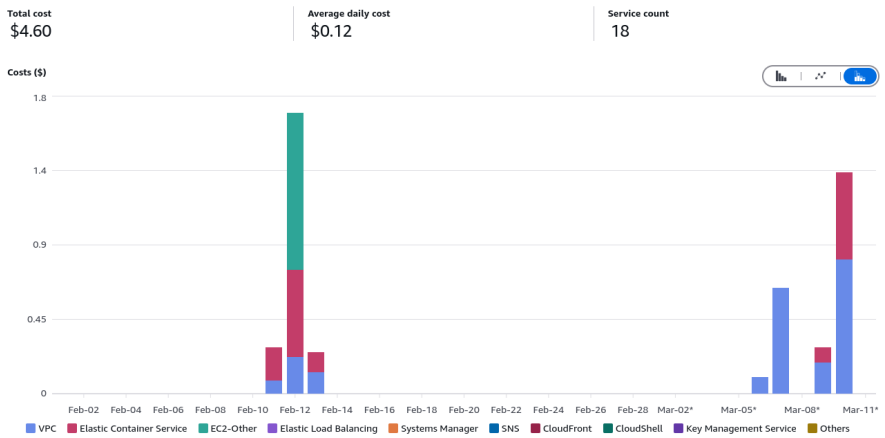
Per quello che riguarda **TypeORM**, Active Record si è dimostrato un pattern valido, sia per sintassi che per performance, se usato con accortezze che includono un uso limitato e possibilmente parallelo di `await` per evitare di bloccare il server ed il caricamento di entità correlate solo quando necessario, per evitare di sovraccaricare la memoria.

In risposta ai movimenti NoSQL, che predicano un'assoluta flessibilità dello schema e del modello dei dati, si può dire che i database relazionali sono ancora una scelta valida per progetti che richiedono una struttura ben definita. Questi, in combinazione con la tipizzazione statica di linguaggi come Typescript consentono ai programmatori di evitare problemi di runtime, catturando errori di sintassi e di tipo già in fase di compilazione, prima ancora che di test. In base a confronti tra coerenza, disponibilità e partizionabilità[4], una soluzione che aggancia TypeORM ad un'istanza PostgreSQL in cloud risulta essere una scelta che bilancia bene questi tre aspetti, privilegiando i primi due.

[4] CAP theorem - Wikipedia.

Gli esperimenti effettuati sul deploy degli applicativi Nuxt-based con **AWS**, hanno mostrato come le architetture serverless, dotate di pool di connessioni a database, forniscono performance comparabili ed in certi casi migliori rispetto a server dedicati, ad un costo di gestione inferiore.

Durante i test effettuati, comprendenti circa 10000 richieste HTTP(s) equamente ripartite tra ECS e Lambda, sono stati rilevati i costi mostrati nel seguente grafico:



Tenuti fissi i costi relativi alla VPC (ammontati a $2.17), e tralasciati i costi dei database RDS e delle imposte, il costo di ECS è stato di $1.47. Il costo di Lambda, approssimato per eccesso è risultato pari a $0.01.

Queste considerazioni però non si possono estendere a tutti i casi d'uso e ad ogni scala. A riguardo, si può citare un articolo di Marcin Kolny[5] che ha descritto la migrazione di alcuni sistemi di analisi dati di Amazon Prime Video da un sistema serverless a uno basato su ECS, che ha consentito di ridurre i costi del 90% e di aumentare le performance. L'overhead aggiunto dalle troppe funzioni Lambda avviate a cascata (sia in dovuto al cold start che al *marshalling* di dati verso altri microservizi) è stato il fattore determinante di questa scelta.

Nonostante questo, AWS Lambda rimane il servizio di esecuzione di codice in cloud che crea il compromesso più competitivo in termini di costo e performance per progetti su scala media e piccola, e rimane un'ottima piattaforma per servire applicazioni Nuxt con API che si interfacciano a database relazionali distribuiti mediante TypeORM. La repository sulla quale sono stati eseguiti i test, disponibile su github.com/iacobucci/cfn-nuxt-typeorm-lambda-aurora, può fare da punto di partenza per ulteriori esperimenti e sviluppi.

Possibili estensioni di questo lavoro vanno in direzioni di:

- Ampliamento dell'infrastruttura cloud delle applicazioni di esempio per includere altri servizi AWS, come S3 per il salvataggio e reperimento di file statici, assieme a CloudFront per la loro distribuzione.
- Utilizzo mirato di strategie di rendering diverse in Nuxt, anche reso possibile servizi AWS aggiuntivi. Un'estensione interessante che si potrebbe apportare al framework stesso potrebbe riguardare finalizzazioni dell'implementazione delle "server functions", come descritte nel paragrafo Endpoint API.

[5] Scaling up the Prime Video audio/video monitoring service and reducing costs by 90% - versione archiviata dell'articolo originale.

- Implementazione di un sistema di cache in memory per le query TypeORM più frequenti, come Redis. Una miglioria che si potrebbe apportare alla libreria stessa potrebbe essere l'estensione dell'API `find()` anche alle colonne di tipo `json` o `jsonb` di database relazionali che le supportano, come PostgreSQL, per ottenere i benefici di una struttura rigida e ben definita assieme alla flessibilità di sistemi di persistenza orientati a documenti.

Bibliografia

Testi di riferimento

Capitolo 1

1. CERN - A short history of the Web - un articolo che percorre velocemente le origini del world wide web.
2. ECMA - 262 - lo standard ufficiale di Javascript.
3. Mozilla developer network web docs - la documentazione e le linee guida ufficiali di Mozilla per lo sviluppo web.
4. A brief history of CSS until 2016 - un articolo che celebra i 20 anni di CSS, spiegando la sua storia e le sue evoluzioni.
5. Honeypot - Node.js: The Documentary - un documentario sulla storia di Node, di NPM, e dei loro ideatori.
6. Honeypot - How a small team of developers created React at Facebook - un documentario sulla storia di React, i primi progetti che lo hanno adottato e la comunità di sviluppatori Open source che lo supporta.
7. Honeypot - Vue.js: The Documentary - un documentario sulla libreria Vue e sulle esigenze che hanno portato alla sua creazione.
8. OfferZen - Typescript Origins: The Documentary - un documentario sulla storia di Typescript, del suo team di sviluppo e selle politiche Open source di Microsoft.
9. Google developers - Core web vitals - una guida di Google sulle metriche di performance web.
10. Web browser engineering - un libro online di Pavel Panchekha e Chris Harrelson che spiega come funzionano i browser web e mostra un'implementazione Python di un browser completo.

Capitolo 2

1. Introduction to Node.js - una guida introduttiva a Node.js.
2. Typescript official documentation - la documentazione ufficiale di Typescript.

3. Nuxt official documentation - la documentazione ufficiale del framework Nuxt.
4. Vue template syntax - la guida ufficiale alla sintassi dei template di Vue.
5. Mozilla HTTP Reference - riferimenti sulle specifiche HTTP.
6. TypeORM official documentation - la documentazione ufficiale della libreria TypeORM.
7. Vite official documentation - la documentazione ufficiale del bundler Vite.
8. Vitest official documentation - la documentazione ufficiale del test runner Vitest.
9. PostgreSQL 17 documentation - la documentazione ufficiale del database PostgreSQL nella versione 17.4.

Capitolo 3

1. AWS Cloudformation Documentation - la documentazione ufficiale di AWS Cloudformation, completa di riferimenti all'API yaml, usata nella trattazione.
2. AWS CloudFormation Starter Workflow for GitHub Actions - un repository GitHub che contiene uno scheletro per realizzare workflow di GitHub Actions che avviano deploy di stack Cloudformation.
3. ISO/IEC 22123-2:2023 - lo standard internazionale per "Information technology - Cloud computing".
4. AWS IAM Documentation - la documentazione ufficiale di AWS Identity and Access Management.
5. AWS ECS Documentation - la documentazione ufficiale di AWS Elastic Container Service.
6. AWS ECR Documentation - la documentazione ufficiale di AWS Elastic Container Registry.
7. AWS Elastic load balancing Documentation - la documentazione ufficiale di AWS Elastic Load Balancing.
8. AWS RDS Documentation - la documentazione ufficiale di AWS Relational Database Service.
9. AWS Lambda Documentation - la documentazione ufficiale di AWS Lambda.
10. AWS RDS Proxy - la documentazione ufficiale per le proxy su AWS RDS.
11. Fireship - il canale YouTube di Jeff Delaney, una risorsa comprensiva delle molte aree dello sviluppo web, sul quale sono pubblicati tutorial, introduzioni a nuove tecnologie e notizie di attualità sul settore informatico.

Ringraziamenti

Ringrazio il mio relatore, il Prof. Paolo Bellavista, per la sua professionalità e per le sue preziose lezioni, assieme ai docenti dei corsi di Ingegneria Informatica. Ringrazio i miei genitori, la mia famiglia ed i miei amici per l'incoraggiamento e il supporto che mi hanno dato durante la stesura di questo lavoro. Un ringraziamento speciale va alla mia cara Aurélie. Infine un ringraziamento a tutti i membri della comunità Open Source che condividono le loro esperienze ed il loro lavoro.

Strumenti Open source utilizzati

- **Code** - la build open source di Visual Studio Code, utilizzata per la scrittura e la correzione del testo.
- **Markdown** - il formato di scrittura utilizzato per la stesura del testo.
- **Latex** - per la formattazione del testo.
- **Pandoc** - per la conversione dei file Markdown in Latex.
- **Minted** - per la colorazione della sintassi dei codici sorgente.
- **Panflute** - per la creazione di filtri personalizzati per Pandoc.
- **Mermaid** - per la creazione di diagrammi UML, ER, di sequenza e di flusso.