PROGETTO TEST 22HBG S.r.l. - BACKEND

Elena Zerbin

26 aprile 2024

Sommario

Affrontare il Test proposto dall'azienda si è rivelato un'esperienza professionale estremamente gratificante e ricca di insegnamenti. Come sviluppatore, l'opportunità di immergermi in linguaggi di programmazione precedentemente inesplorati ha rappresentato una sfida entusiasmante che ha stimolato la mia passione per l'apprendimento e la crescita tecnica.

Questo progetto non è stato solo una prova delle mie abilità tecniche ma anche una finestra aperta su nuovi orizzonti tecnologici. Mi ha spinto a investigare e risolvere problemi complessi, un esercizio che si è dimostrato tanto impegnativo quanto formativo. Confrontarmi con situazioni sconosciute e cercare soluzioni efficaci ha rappresentato l'essenza stessa del problem-solving, un'abilità fondamentale nel mio campo.

La conoscenza di nuove tecnologie e la loro implementazione pratica mi hanno consentito di espandere il mio toolkit di sviluppo e di vedere in azione concetti teorici. Inoltre, il progetto mi ha permesso di trarre da precedenti esperienze lavorative e di studi, consolidando e applicando la conoscenza in un contesto reale e dinamico.

In questa relazione, descriverò il processo di sviluppo del web service API RESTful, evidenziando le scelte tecniche, le sfide incontrate e le soluzioni adottate. Sarà una riflessione sul percorso che ho intrapreso, sugli ostacoli superati e sulle competenze acquisite durante questa avventura tecnologica.

1 Pianificazione e Ricerca Iniziale

L'avvio di qualsiasi progetto richiede un'attenta fase di pianificazione e ricerca. Consapevole dell'importanza di questa fase preliminare, ho iniziato il mio percorso esplorando la vasta documentazione disponibile su TypeScript. La mia intenzione era quella di comprendere a fondo le caratteristiche distintive di TypeScript rispetto a JavaScript, in particolare la sua natura tipizzata che migliora la sicurezza e la manutenibilità del codice.

La visione di progetti esistenti si è rivelata cruciale: analizzando codice TypeScript reale e applicazioni ben strutturate, sono stata in grado di apprezzare l'efficacia di questo linguaggio nel contesto dello sviluppo di applicazioni scalabili e robuste. L'osservazione diretta di codice altrui (tramite github e Medium) ha fornito preziosi spunti su come organizzare il progetto, strutturare i file, gestire le dipendenze e definire le interfacce e i tipi per una documentazione autoconsistente.

Il passo successivo è stato quello di determinare gli strumenti e i pacchetti necessari per eseguire file TypeScript e per sviluppare un web service in Node.js utilizzando TypeScript ed Express.js. Questo mi ha portato a comprendere il ruolo vitale di ciascuno dei seguenti componenti nel nostro stack tecnologico:

- **Node.js:** Il motore di esecuzione JavaScript lato server che funge da fondamento per il nostro web service, permettendo di costruire applicazioni veloci e scalabili.
- **TypeScript:** Un sovrainsieme di JavaScript che offre vantaggi significativi in termini di tipizzazione statica e che, alla fine, si compila in JavaScript puro, assicurando la compatibilità con Node.js.
- **Express.js:** Il framework di Node.js per la costruzione di applicazioni web, scelto per la sua leggerezza, flessibilità e vasto ecosistema di middleware, che facilita lo sviluppo di API RESTful.

Con le conoscenze acquisite e gli strumenti selezionati, mi sono sentita pronta a tuffarmi nello sviluppo, pronta ad affrontare con fiducia la costruzione del web service.

2 Sviluppo del Progetto

Il processo di sviluppo del web service è stato metodico e strutturato, iniziando dalla configurazione dell'ambiente di sviluppo fino alla stesura del codice. Ho scelto Visual Studio Code (VS Code) come ambiente di sviluppo integrato (IDE) per il suo supporto esteso per TypeScript e le sue funzionalità intelligenti di debugging e gestione del codice, che hanno reso lo sviluppo più efficiente e meno soggetto a errori.

2.1 Configurazione Iniziale

Il primo passo pratico è stato la creazione di un nuovo repository su GitHub. Questo non solo mi ha permesso di tracciare le modifiche e gestire il versionamento del codice, ma anche di esplorare le potenzialità di collaborazione e condivisione di GitHub. Dopo aver creato il repository, l'ho clonato sul mio sistema locale per iniziare lo sviluppo.

2.2 Installazione dei Pacchetti

Prima di immergermi nella codifica, ho configurato l'ambiente di sviluppo installando i pacchetti necessari. Utilizzando npm, il gestore di pacchetti per Node.js, ho installato TypeScript, Express.js e altri middleware essenziali.

2.3 Analisi e Pianificazione

Con l'ambiente pronto, ho dedicato del tempo all'analisi del brief del progetto e del file fornito con il link. Questo passo era cruciale per comprendere a fondo lo scopo e gli obiettivi del web service, assicurando che ogni feature richiesta fosse chiaramente definita e compresa prima di iniziare a scrivere il codice.

2.4 Codifica

Il passaggio alla codifica è stato guidato da una combinazione di esplorazione autonoma e apprendimento da esempi esistenti. Ho cercato progetti simili che utilizzavano TypeScript e Express.js per vedere come altri sviluppatori avevano implementato funzionalità simili. Questo non solo ha aiutato a solidificare la mia comprensione della sintassi e delle strutture di TypeScript,

ma ha anche fornito un riferimento pratico su come strutturare il codice in modo efficiente e mantenibile.

2.5 Sviluppo Iterativo

L'approccio allo sviluppo è stato iterativo. Ho iniziato con la creazione delle basi dell'applicazione, definendo semplici route e la logica di base, e progressivamente ho integrato funzionalità più complesse come la gestione della cache e l'interazione con il database. Ogni step è stato testato e verificato per garantire che soddisfacesse i requisiti specificati.

3 Sviluppo del Sistema di Gestione della Cache

Integrare un sistema efficace di gestione della cache è stata una delle priorità del nostro progetto API, per ridurre il carico sul server e migliorare le prestazioni complessive dell'applicazione. Ho deciso di implementare una soluzione di caching sfruttando le capacità di Redis, un database in memoria noto per la sua velocità e efficienza nel gestire grandi volumi di dati in modo transitorio.

Ho scelto di esplorare sia la soluzione di caching nativa con un oggetto JavaScript in Express.js sia l'implementazione con Redis per diversi motivi:

- Apprendimento e Confronto: Implementando entrambe le tecniche, avevo l'opportunità di valutare direttamente le differenze in termini di prestazioni, scalabilità e facilità di gestione.
- Adattabilità: Utilizzare due approcci differenti mi ha permesso di comprendere meglio quale soluzione fosse più adatta a seconda delle diverse esigenze di caching e della complessità delle query.
- Riserva di Affidabilità: Avere un sistema di caching alternativo implementato fornisce una riserva di affidabilità, permettendo di passare all'altro sistema in caso di necessità o malfunzionamenti.

3.1 Implementazione della Cache con un Oggetto JavaScript in Express.js

- **Configurazione della Cache:** Inizialmente, ho creato un oggetto JavaScript per funzionare come una semplice cache in memoria. Questo oggetto è utilizzato per memorizzare le risposte delle API sotto forma di coppie chiave-valore, dove la chiave è un identificatore unico per ogni richiesta (ad esempio, combinando il titolo e il numero di post richiesti).

- **Integrazione nell'API:** Nella logica dell'API, prima di procedere con il recupero dei dati dal database, il sistema verifica se la risposta per la richiesta specifica è già presente nella cache. Se i dati sono disponibili, vengono restituiti direttamente dalla cache, bypassando così il database e riducendo i tempi di risposta.

- **Gestione dell'Invalidamento della Cache:** Per assicurare che i dati restituiti siano sempre aggiornati, ho implementato una strategia di invalidamento della cache. Ogni volta che i dati rilevanti vengono modificati nel database, le voci corrispondenti nella cache vengono invalidate o aggiornate. Questo assicura che eventuali richieste future riflettano le modifiche più recenti.

3.2 Motivazioni per l'Implementazione della Soluzione di Caching Interna

L'adozione di questo approccio interno per la gestione della cache era motivata da diversi fattori:

- Semplicità e Controllo: Utilizzare un oggetto JavaScript per la cache offre una soluzione semplice e immediata, che è facile da implementare e personalizzare secondo le esigenze specifiche del progetto.

- Costi Ridotti: Non richiedendo infrastrutture esterne o servizi aggiuntivi, questa soluzione riduce i costi operativi associati al mantenimento di strumenti come Redis.

- Rapidità di Implementazione: La velocità con cui si può mettere in pratica questa soluzione permette di accelerare lo sviluppo, particolarmente utile in fasi iniziali del progetto o in ambienti di testing.

3.3 Implementazione della Cache con Redis

- Installazione e Configurazione di Redis: Per iniziare, ho installato Redis sul server utilizzando il pacchetto redis per Node.js. Questo può essere facilmente aggiunto tramite npm con il comando npm install redis.

- Integrazione con Express.js: Utilizzando la libreria client Redis per Node.js, ho integrato Redis nel nostro ambiente Express.js. Questo setup permette di intercettare le richieste API e verificare se una risposta valida è già disponibile in cache prima di procedere con ulteriori elaborazioni o chiamate al database.

- Gestione delle Richieste API: Specificatamente per l'endpoint GET /posts-filtered?title=<testo da ricercare nel titolo>&items=<numero di post da ritornare>, il sistema verifica prima la presenza in cache dei dati richiesti. Se i dati sono disponibili, vengono restituiti immediatamente al client, evitando così chiamate superflue al server remoto.

- Ciclo di Aggiornamento dei Post: È stata implementata una strategia per invalidare la cache ogni volta che i post vengono aggiornati. Questo assicura che le richieste successive all'endpoint restituino sempre i dati più aggiornati. La logica di invalidamento della cache è essenziale per mantenere la coerenza dei dati tra la cache e la fonte originale.

3.4 Motivazioni per la Scelta della Doppia Soluzione di Caching

Ho scelto di esplorare sia la soluzione di caching nativa con un oggetto JavaScript in Express.js sia l'implementazione con Redis per diversi motivi:

- Apprendimento e Confronto: Implementando entrambe le tecniche, avevo l'opportunità di valutare direttamente le differenze in termini di prestazioni, scalabilità e facilità di gestione.

- Adattabilità: Utilizzare due approcci differenti mi ha permesso di comprendere meglio quale soluzione fosse più adatta a seconda delle diverse esigenze di caching e della complessità delle query.

- Riserva di Affidabilità: Avere un sistema di caching alternativo implementato fornisce una riserva di affidabilità, permettendo di passare all'altro sistema in caso di necessità o malfunzionamenti.

3.5 Confronto e Decisione

Avere sia la cache interna con un oggetto JavaScript sia Redis mi ha permesso di valutare le prestazioni e l'efficacia di entrambe le soluzioni in scenari reali. Questo confronto diretto è stato fondamentale per determinare quale soluzione fosse più adeguata in base alle diverse circostanze e requisiti di carico del sistema.

In conclusione, l'integrazione di queste due soluzioni di caching ha arricchito il progetto, migliorando le prestazioni e fornendo flessibilità nella gestione dei dati. Questa esperienza ha anche ampliato la mia comprensione delle tecniche di caching e della loro applicazione pratica in progetti software.

4 Scelta del Database

Il processo di selezione del database per il nostro progetto è stato un elemento cruciale dello sviluppo, riflettendo un approccio metodico e informato. Inizialmente familiare solo con MySQL Workbench, ho deciso di esplorare altre opzioni per assicurare una scelta ben ponderata.

Dopo un'analisi dettagliata, ho optato per PostgreSQL per il database del mio progetto, motivato da diverse considerazioni chiave:

- **Affidabilità:** PostgreSQL offre supporto completo per le transazioni ACID, garantendo l'affidabilità delle operazioni e l'integrità dei dati, elementi essenziali per applicazioni dove la sicurezza dei dati è prioritaria.

- **Scalabilità:** La capacità di gestire grandi volumi di dati e numerosi utenti simultanei senza perdere in performance rende PostgreSQL ideale per supportare applicazioni in rapida espansione.

- **Flessibilità:** La compatibilità con un'ampia varietà di tipi di dati, inclusi JSON per dati non strutturati e dati spaziali per sistemi di informazione geografica (GIS), permette a PostgreSQL di adattarsi a diverse esigenze applicative.

La scelta di PostgreSQL è stata dunque basata non solo sulla mia precedente esperienza con altri sistemi di gestione del database, ma principalmente sulle sue superiori capacità di soddisfare le complesse esigenze del progetto in modo efficace e affidabile.

5 Discussione

Durante lo sviluppo del progetto, ho incontrato alcune difficoltà significative in due aree principali: l'utilizzo di Swagger per la documentazione delle API e l'implementazione di un sistema di caching interno. Ecco come ho affrontato e superato queste sfide:

5.1 Difficoltà con Swagger

L'adozione di **Swagger** rappresentava una novità assoluta per me. Inizialmente, non ero familiare con questo strumento e non sapevo come gestire la documentazione delle API in modo efficace. Tuttavia, dopo qualche ora di ricerca approfondita e sfruttando il mio intuito, sono riuscita a comprendere le funzionalità di Swagger e a utilizzarlo per sviluppare una documentazione chiara e dettagliata delle API. Questo strumento si è rivelato estremamente utile per definire gli endpoint, i parametri e i modelli di risposta, migliorando significativamente l'interoperabilità e la manutenibilità dell'API.

5.2 Problemi nell'Implementazione della Cache Interna

Per quanto riguarda il sistema di caching interno, ho implementato inizialmente un middleware di cache che sfruttava un semplice oggetto JavaScript per memorizzare le risposte delle API. Questa implementazione presentava tuttavia alcuni problemi:

- Sovrascrittura dei Metodi di Risposta: Il tentativo di sovrascrivere direttamente res.send o res.json per inserire i dati nella cache ha creato problemi di compatibilità e funzionalità, poiché questi metodi hanno firme e comportamenti specifici che devono essere mantenuti. La sovrascrittura diretta può portare a comportamenti imprevisti e bug difficili da tracciare.

- Persistenza e Unicità delle Chiavi: La mia cache in memoria non manteneva i dati tra i riavvii del server, e la chiave di cache basata su req.originalUrl non sempre garantiva l'unicità per diverse combinazioni di parametri di query. Questo poteva portare a collisioni o a errori nel recupero dei dati.

5.3 Soluzione Migliorata per il Caching

Per risolvere questi problemi, ho apportato le seguenti modifiche al sistema di caching:

- Intercettazione del Flusso di Risposta: Invece di sovrascrivere i metodi res.send o res.json, ho optato per un approccio che intercetta il flusso di dati della risposta prima che questa venga inviata al client. Questo mi ha permesso di catturare e memorizzare i dati in cache senza alterare il comportamento standard della risposta di Express.

- Gestione Avanzata delle Chiavi di Cache: Ho migliorato la gestione delle chiavi di cache assicurandomi che fossero uniche per ogni combinazione di parametri di query, evitando così il rischio di collisioni o dati errati.

- Implementazione del TTL (Time To Live): Ho aggiunto un timestamp di scadenza per ogni elemento in cache, permettendo così di invalidare automaticamente i dati vecchi o obsoleti dopo un certo periodo di tempo, garantendo che i dati restituiti siano sempre aggiornati e validi.