Backend con JS - Appunti 18-02-26

Stack Tecnologico e Architettura Backend Java

Lo stack tecnologico rappresenta l'insieme completo degli strumenti necessari per realizzare un prodotto software. Nel contesto dello sviluppo backend Java tradizionale, lo stack classico si compone di MySQL come database relazionale, Spring Boot come framework applicativo principale, Spring Data JPA come livello di accesso ai dati — che si appoggia su Hibernate per la mappatura oggetto-relazionale — e Spring Web per esporre le API REST.

L'architettura che ne deriva è stratificata su quattro livelli distinti. Alla base si trova il database MySQL, responsabile della persistenza fisica dei dati. Sopra di esso opera il livello Repository, implementato tramite Spring Data JPA, che gestisce le operazioni CRUD e le query verso il database astraendo i dettagli dell'accesso ai dati. Il livello successivo è quello dei Service, realizzato con Spring puro, che contiene tutta la logica di business e orchestra le operazioni tra diversi repository. In cima si trova il livello API, costruito con Spring Web, che espone i controller REST attraverso cui il frontend invia richieste HTTP e riceve risposte. Di questi quattro livelli, tre sono scritti in Java mentre uno è puramente SQL.

I Due Approcci allo Sviluppo Web Java

Quando si sviluppa un'applicazione web Java esistono due approcci principali con filosofie diverse. Il modo migliore dal punto di vista formativo consiste nell'installare Tomcat come server locale sulla propria macchina. Tomcat resta in ascolto sulla porta 8080, quindi digitando `http://localhost:8080` nel browser ci si collega direttamente al server in esecuzione locale. All'interno di Tomcat l'applicazione viene organizzata rispettando la separazione dei livelli: le API gestiscono l'ingresso e l'uscita dei dati, i Service contengono la logica applicativa, il Repository dialoga con MySQL. Questa struttura costringe a ragionare fin da

subito secondo i principi architetturali utilizzati nelle applicazioni professionali, dove ogni componente ha responsabilità ben definite e la manutenzione del codice risulta più semplice grazie alla modularità.

Il modo rapido, invece, bypassa molti passaggi usando tecnologie come Express e Better SQLite. Express è un framework JavaScript per Node.js che permette di creare un server web funzionante con poche righe di codice, mentre Better SQLite è un wrapper per SQLite che consente di avere un database funzionante senza installazioni esterne, semplicemente leggendo e scrivendo su un file locale. Questo approccio permette di scrivere direttamente il codice che gestisce le richieste HTTP senza dover configurare server separati o orchestrare livelli distinti, consentendo di vedere rapidamente un'applicazione funzionante ma nascondendo i dettagli architetturali che emergono invece con l'approccio tradizionale basato su Tomcat.

Il Contratto HTTP e l'Indipendenza tra Frontend e Backend

Un principio fondamentale dell'architettura web è che le API e il frontend non devono necessariamente essere scritti nello stesso linguaggio. Il client che consuma le API non ha alcuna consapevolezza di cosa ci sia nel server: potrebbe essere Java con Spring, JavaScript con Express, Python con Django o qualsiasi altra tecnologia. L'unica cosa che conta è il contratto HTTP, ovvero che il server risponda correttamente alle richieste secondo il formato concordato.

Nel contesto dello stack rapido basato su Node.js, Express rappresenta l'alternativa a Tomcat combinato con Spring Web per la gestione delle richieste HTTP. Better SQLite sostituisce l'intero stack composto da MySQL e Spring Data JPA, offrendo un database relazionale leggero basato su file senza necessità di installare un server separato. Il modulo CORS serve invece per permettere al frontend di comunicare con il backend superando le restrizioni di sicurezza del browser relative alle richieste cross-origin.

L'istanza del server Express viene creata invocando la funzione `express()` e assegnandola alla costante `app`. Questa singola istruzione sostituisce tutta la configurazione che in un progetto Spring Boot richiederebbe la definizione di controller, service e repository distribuiti su classi separate. Con Express il server

è già pronto per ricevere richieste HTTP e gestirle tramite rotte definite direttamente nell'applicazione, senza la necessità di stratificare i livelli secondo il pattern architetturale tipico delle applicazioni enterprise Java.

NPM è il package manager di Node.js e si occupa di scaricare le dipendenze necessarie al progetto e di configurarlo correttamente. Node.js è uno strumento estremamente versatile che permette di fare praticamente qualsiasi cosa nel mondo JavaScript: può essere usato per scrivere API backend, per installare Angular CLI e gestire progetti frontend, o per eseguire qualsiasi altro tipo di applicazione JavaScript lato server.

Testing delle API e Programmazione Asincrona

Un'API non può essere validata affidandosi esclusivamente a prove manuali. Le prove manuali sono utili durante lo sviluppo, ma non si manda in produzione senza test automatici. Scrivere un test significa scrivere un programma JavaScript che chiamerà l'API su un determinato endpoint e ne controllerà il risultato.

La funzione `testAPI` viene dichiarata come `async` perché deve gestire operazioni asincrone, ovvero operazioni che richiedono tempo per completarsi e non bloccano l'esecuzione del programma mentre aspettano una risposta. L'esecuzione non procede in modo lineare: quando viene invocata `fetch(url)`, la richiesta HTTP parte verso il server ma il programma non si ferma ad aspettare. La parola chiave `await` davanti a `fetch` dice al runtime di sospendere l'esecuzione della funzione finché la risposta non arriva dal server, e solo allora la variabile `response` viene popolata con il risultato. Questo meccanismo permette di scrivere codice asincrono che sembra sincrono, evitando di bloccare l'intera applicazione durante le attese. La chiamata `response.json()` è anch'essa asincrona perché deve leggere e parsare il corpo della risposta HTTP, operazione che potrebbe richiedere tempo se il payload è grande. Anche qui `await` sospende l'esecuzione finché la conversione in oggetto JavaScript non è completata.

La Classe Test e il Concetto di Callback

Un test è un oggetto come qualsiasi altra entità nel codice, e in quanto oggetto è rappresentato da una classe. La classe `Test` è pensata per rappresentare un singolo caso di test su un'API. Per testare un'API occorre sapere due cose: cosa si manda e cosa ci si aspetta di ricevere. La parte REQUEST descrive la chiamata da effettuare — l'URL dell'API da colpire, il verbo HTTP (GET, POST, PUT, DELETE) e il body, ovvero i dati eventualmente allegati alla richiesta. La parte RESPONSE descrive invece cosa ci si aspetta di ricevere: lo `statusCode` è il codice numerico di risposta HTTP (200 per successo, 404 per non trovato, 500 per errore del server), mentre `verify` è una funzione che viene applicata al corpo della risposta per controllare che i dati ricevuti siano effettivamente quelli attesi. Ogni istanza di questa classe rappresenta un test del tipo: "chiamo quest'URL con questo verbo e questi dati, e mi aspetto questo codice di risposta con un corpo che supera questo controllo". È un pattern molto comune nei framework di testing automatico delle API.

Passare una funzione come parametro prende il nome di callback. Si chiama callback perché letteralmente significa "richiama indietro": l'idea è che la funzione non viene eseguita subito, ma viene passata a qualcun altro che la richiamerà quando avrà terminato il proprio lavoro. In JavaScript le funzioni sono oggetti come tutti gli altri, quindi possono essere passate come parametri esattamente come si passerebbe un numero o una stringa. Questo consente di scrivere codice flessibile: invece di decidere in anticipo cosa fare con il risultato, si lascia che sia chi chiama il metodo a definire la logica di verifica.

`fetch` è la funzione che si utilizza per effettuare una chiamata HTTP in JavaScript. Le si passa l'URL e opzionalmente le opzioni della richiesta — verbo, body, headers — e la chiamata parte verso il server. Poiché la risposta non arriva istantaneamente, JavaScript non si ferma ad aspettare ma continua ad eseguire il resto del codice. Per questo motivo `fetch` non restituisce direttamente la risposta, ma restituisce una Promise, che è sostanzialmente una promessa: "non ho ancora il risultato, ma quando ce l'ho te lo do". `.then` è il modo in cui si dice alla Promise cosa fare quando la promessa viene mantenuta, ovvero quando la risposta arriva. Gli si passa una callback che verrà eseguita automaticamente nel momento in cui il risultato è disponibile.