APPUNTI-10-02-2026 Progetto WebClinic

Capitolo 1: Cosa sono le API e perchè usare REST

Le API rappresentano il punto di accesso al backend, il canale attraverso cui è possibile interagire con i dati e le funzionalità disponibili. Quando comunicano, usano il formato JSON, che è diventato lo standard per lo scambio di dati in internet. Una buona API deve essere solida e affidabile, non deve mai andare in crash, e soprattutto deve restituire risposte chiare che chi la chiama possa capire facilmente.

Il concetto di REST non è una tecnologia specifica, ma piuttosto uno stile architetturale, cioè un modo di organizzare le cose. Si tratta di una filosofia, un insieme di regole che rendono le API coerenti e intuitive. Le API scritte fino ad ora seguono questo approccio.

L'idea fondamentale di REST è esporre risorse online e ragionare in termini di oggetti. Un indirizzo REST è semplicemente l'indirizzo dove trovare un oggetto specifico sulla rete. Il primo pilastro di REST è che l'indirizzo stesso identifica la risorsa, come ad esempio /patients o /patients/1. Guardando solo l'indirizzo si capisce già di cosa si sta parlando, senza bisogno di altre spiegazioni.

Il verbo HTTP utilizzato è altrettanto importante, perché comunica l'azione che si vuole compiere. GET /patients significa che si vuole leggere tutti i pazienti. POST /patients indica che si sta creando un nuovo paziente. GET /patients/1 significa che si sta leggendo solo il paziente con ID 1. PUT /patients/1 indica che si sta aggiornando quel paziente specifico. Questa combinazione di indirizzi e verbi rende le API REST molto intuitive e facili da capire anche per chi le usa per la prima volta.

Capitolo 2: Gli endpoint e i codici di stato

Un endpoint è un indirizzo legato a un metodo HTTP specifico. Quando si fa una richiesta get /patients/{id}, ci si aspetta di ricevere un paziente. Se tutto va bene, la risposta ha uno status 200, che è il codice HTTP che significa "tutto OK". In

questo caso vengono restituiti i dati del paziente, anche se non necessariamente tutti, solo quelli che si è deciso di includere.

Se le cose vanno male, si può ricevere uno status 404, che significa che il paziente con quell'ID non esiste nel database. Questo fa parte della famiglia 400, che contiene tutti gli errori relativi alla richiesta o al client. Se c'è qualcosa di sbagliato nella richiesta stessa, come dati mancanti o formati errati, si riceve un 400, che sta per "bad request". Questi codici di stato sono fondamentali perché comunicano immediatamente cosa è successo senza dover leggere il corpo della risposta.

Capitolo 3: Primi passi con il progetto WebClinic

Il progetto si chiama webclinic e fa parte del package com.generation.webclinic. Per cominciare, si configura il file application properties, prendendo ispirazione da un file già scritto che funziona correttamente. Questo fa risparmiare tempo e assicura di avere una base solida.

A questo punto si crea il database, ma senza creare le tabelle manualmente. Ci pensa Hibernate, l'ORM che mapperà automaticamente le entità in tabelle del database. Questo è uno dei grandi vantaggi di usare Spring Boot: la configurazione del database diventa quasi automatica.

Capitolo 4: Definire l'entità Paziente

Il prossimo passo è creare l'entità Paziente nella cartella model.entities. Questa entità rappresenta il paziente nel sistema e mapperà una tabella nel database. Si definiscono tutti i campi che servono per identificare un paziente in modo univoco.

La struttura base prevede un ID autogenerato con strategy identity, poi firstName per il nome, lastName per il cognome, ssn per il codice fiscale, gender per il genere, dob per la data di nascita, address per l'indirizzo e city per la città. Queste informazioni sono sufficienti per identificare ogni paziente nel sistema. Spring vede solo i getter e i setter, non le proprietà dirette, quindi devono essere generati tutti.

Si aggiunge anche un campo history, che rappresenta la storia clinica del paziente, dove poter annotare visite, esami e altre informazioni mediche importanti.

Capitolo 5: Il problema della validazione ripetitiva

Man mano che il progetto cresce, ci si rende conto di dover validare gli stessi dati in posti diversi. L'email, ad esempio, va controllata spesso: quando si registra un paziente, quando si modifica un dottore, quando si crea un account utente. Sarebbe stupido riscrivere la stessa logica di validazione ogni volta.

La soluzione elegante è creare un servizio condiviso che può essere iniettato ovunque serva. Questo oggetto risiederà nel context di Spring e sarà accessibile a tutti i componenti che ne hanno bisogno. In questo modo si centralizza la logica di validazione e la si riutilizza in tutto il progetto, mantenendo il codice pulito e DRY (Don't Repeat Yourself).

Capitolo 6: CommonValidator, un servizio condiviso

Si crea una nuova classe chiamata CommonValidator nel package com.generation.webclinic.model.entities. Questo validatore è utile a tutte le entità del progetto, non solo a Patient. Si usa l'annotazione @Service per trasformarlo in un bean Spring, che automaticamente viene messo a disposizione nel context dell'applicazione.

All'interno di questo servizio si preparano due metodi principali. Il primo usa una regex per validare il Codice Fiscale Italiano, che ha un formato molto specifico: sei lettere per il nome, due numeri, una lettera per il mese, due numeri, una lettera per il giorno, due numeri finali e una lettera di controllo. Il secondo metodo valida il formato dell'email usando un pattern standard.

Il metodo validEmail controlla che l'email non sia nulla o vuota e poi verifica che rispetti il pattern. Stessa cosa per validSSN, che lavora sul codice fiscale in maiuscolo. Questi metodi torneranno utili ovunque nel progetto.

Capitolo 7: L'interfaccia Validable

Per rendere il sistema ancora più pulito, si crea un'interfaccia chiamata Validable. Questa interfaccia definisce un contratto: qualsiasi classe che la implementa deve fornire un metodo getErrors() che restituisce una lista di stringhe con gli errori di validazione.

Si aggiunge anche un metodo concreto chiamato isValid(), che semplicemente restituisce true se la lista di errori è vuota. Questo permette di controllare

rapidamente se un oggetto è valido senza dover esaminare tutti gli errori uno per uno.

Nel CommonValidator si aggiunge anche un metodo helper chiamato emptyString() che controlla se una stringa è nulla o contiene solo spazi bianchi. Patient deve implementare Validable, quindi può essere validato in modo uniforme rispetto ad altre entità.

Capitolo 8: Il modificatore transient e il repository

Quando si chiama getErrors(), si verifica lo stato dell'entità indipendentemente da dove arrivi. Il modificatore transient è importante in Java: indica che quella proprietà non deve essere serializzata. Nel caso in questione, la lista degli errori non deve andare nel database, non deve essere serializzata nelle comunicazioni di rete e non deve essere mandata al client.

Si prepara poi il repository per Patient, che permette di eseguire operazioni CRUD sul database. Il repository è l'interfaccia tra il codice e il database, nascondendo la complessità delle query.

Capitolo 9: Perchè usare i DTO invece delle entità

Quando si progettano le API, si deve fare una scelta importante: restituire l'entità completa o creare un oggetto diverso per la comunicazione con il client? La risposta è quasi sempre la seconda. L'entità è il modo con cui si parla con il database, con il client si parla tramite DTO, Data Transfer Object.

Perché questa distinzione? Perché si potrebbe voler mandare dati diversi a seconda di chi fa la richiesta. La storia clinica completa dovrebbe essere visibile solo ai dottori, mentre i dati anagrafici potrebbero essere visibili anche ai segretari. È possibile creare diverse mappature per la stessa entità e restituire DTO diversi in base alle esigenze.

Inoltre, non è detto che si debba sempre restituire qualcosa. Se l'ID richiesto non esiste, si restituisce solo uno status 404 senza corpo, perché non ha senso mandare dati vuoti. Questo rende le risposte più efficienti e chiare.

Capitolo 10: Implementare l'endpoint GET

Si implementa il metodo findPatient che risponde alle richieste GET per /patients/{id}. Si usa @GetMapping per indicare che questo metodo gestisce le

richieste di lettura. Il metodo restituisce ResponseEntity, che permette di controllare esattamente cosa mandare indietro, incluso lo status HTTP.

Il parametro @PathVariable("id") collega l'ID nell'URL al parametro del metodo. Si cerca il paziente nel repository usando patientRepo.findById(id), che restituisce un Optional. Se l'Optional è vuoto, significa che il paziente non esiste, quindi si restituisce 404 con notFound().build(). Se esiste, si recupera il paziente dall'Optional e si prosegue.

Questo approccio segue il principio REST di restituire solo lo status corretto quando la risorsa non viene trovata, senza inviare corpi di risposta inutili.

Capitolo 11: Anatomia di una risposta HTTP

Una risposta HTTP è composta da due parti principali: gli headers e il body. Gli headers sono metadati sulla risposta, come il tipo di contenuto, la lunghezza, e soprattutto lo status. Il body è il corpo effettivo della risposta, che contiene i dati, ma è opzionale.

Lo status è forse l'header più importante. Comunica immediatamente al client cosa è successo: se la richiesta è andata a buon fine (200), se il contenuto non è stato trovato (404), se c'è un errore del server (500), e così via. Non serve leggere il body per capire il risultato della richiesta, lo status basta.

Capitolo 12: Il pattern Singleton per CommonValidator

Per assicurarsi che esista una sola istanza di CommonValidator in tutta l'applicazione, si implementa il pattern Singleton. Il costruttore è privato, quindi nessun altro può creare istanze. La variabile statica instance contiene l'unica istanza, creata quando la classe viene caricata.

Il metodo getInstance() è il punto di accesso globale: chiunque vuole usare CommonValidator deve passare da qui e riceverà sempre la stessa istanza. Questo ha senso per un validatore, perché si vuole che le regex e le regole di validazione siano condivise, non che ogni modulo abbia la sua copia.

L'annotazione @Service dice a Spring che questa classe appartiene al layer di servizio e deve essere gestita dal container IoC. Spring scansionerà il classpath, troverà questa classe e la renderà disponibile per l'autowiring.

Capitolo 13: Creare il PatientDTO

Si implementa PatientDTO, che è l'oggetto che viene usato per comunicare con il client, non con il database. Il DTO è basato sull'entità ma può essere personalizzato. Ad esempio, si sceglie di non riempire i campi errori e valid, perché non servono nelle risposte al client.

La cosa bella dei DTO è che si possono creare quante mappature si vuole per la stessa entità. Si può avere un BasicDTO con solo i campi essenziali, un FullDTO con tutti i dati, e magari un MedicalDTO con solo le informazioni cliniche. Ogni DTO serve uno scopo specifico.

Capitolo 14: Il Mapper e la conversione tra oggetti

Si scrive il mapper che converte da Patient a PatientDTO. È un processo semplice: si crea un nuovo oggetto DTO e si copiano tutti i dati che servono dall'entità. Si può decidere quali campi includere e quali escludere, trasformarli se necessario, o aggiungerne di nuovi.

Si implementano due versioni: toBasicDTO che include solo i dati essenziali, e toFullDTO che include anche la lista degli errori. Quest'ultimo è utile quando si vuole comunicare al client non solo i dati ma anche eventuali problemi di validazione. Il Service userà uno o l'altro a seconda delle esigenze specifiche.

Questa separazione tra entità e DTO rende il codice molto flessibile: si può cambiare la struttura del database senza toccare le API, e si può cambiare cosa si manda ai client senza toccare il modello dati.

Capitolo 15: Considerazioni sulle performance

Il tempo della request si misura in millisecondi. Se si devono fare 3 request sequenziali, rischiano di durare 3 secondi, che è un tempo troppo lungo per molte applicazioni. Per questo motivo, quando si progettano le API, bisogna sempre considerare il numero di chiamate necessarie e cercare di minimizzarle, ad esempio aggregando i dati in un'unica risposta quando possibile.