Riunione con IMOLA INFORMATICA

RAMtastic6

10 maggio 2024

email: ramtastic6@gmail.com

Informazioni sul documento

Redattori:	Filippo T. Samuele V. Riccardo Z.
Verificatori:	Riccardo Z. Samuele V.

Verificatori esterni: 

Indice

1 Informazioni sulla riunione

- **Luogo**: Microsoft Teams;
- **Ora di inizio**: 12:00;
- **Ora di fine**: 13:00;
- **Partecipanti**:

NOME	DURATA
Alessandro (IMOLA INFORMATICA)	1o
Filippo T.	45m
Leonardo B.	1o
Michele Z.	1o
Riccardo Z.	1o
Samuele V.	1o
Davide B.	1o

Tutti i membri si sono collegati alle 12:00. Filippo T. è dovuto uscire 15 minuti prima per impegni personali.

2 Ordine del giorno

- Feedback sullo schema della progettazione;
- Consigli lato architettura/design pattern;
- Come implementare notifiche (socket);
- Riscontro unit test (backend);
- Dubbi sul flusso dell'applicazione (serverAction - nestjs);
- Come validare dati lato backend.

3 Riassunto dell'incontro

3.1 Feedback sullo schema della progettazione

L'incontro si è aperto con l'esposizione tramite condivisione schermo di una prima versione dello schema relativo alla progettazione del prodotto software.
Il feedback da parte del proponente è stato positivo riguardo la struttura dello schema. Alessandro si è però soffermato sull'attuale rappresentazione della parte relativa al Socket. E' stato consigliato al gruppo di migliorarla anche tramite descrizione testuale dei singoli componenti, in quanto, per come esposta al momento dell'incontro, non risultava chiaro come venisse gestito il flusso della Socket.

3.2 Connessione al database tramite le *Actions* di *NextJS*

La seconda domanda posta dal gruppo al proponente è stata relativa all'attuale implementazione della funzionalità di *login*. E' stato esposto che al momento viene usata il tool, fornito dal framework *NextJS*, delle *Actions*.
E' stato spiegato dal proponente che l'attuale funzionamento è corretto, in quanto il *tool* si prende a carico anche del meccanismo di *caching*, che rende meno oneroso il quantitativo di risorse richiesto lato client, andando dunque a pesare meno, in termini di *performance*, su quest'ultimo.

3.3 Gestione degli errori nella validazione degli input (backend)

Proseguendo con l'incontro, un'altra domanda posta dal gruppo ha riguardato l'attuale funzionamento del "*Controller*": allo stato descritto, infatti, il "*Controller*" è responsabile di andare a chiamare i metodi sul "*Service*", il quale fa il controllo degli input al momento del *login*.
La risposta di Alessandro è stata che la verifica dell'input deve avvenire sul "*Controller*, in quanto il *framework* usato per il *backend*, *NEST*, mette a disposizione dei *tools* con dei *decorators* apposta per fare la *validation*, qui di seguito vi è il link alla documentazione che spiega come fare: https://docs.nestjs.com/techniques/validation.

3.4 Implementazione degli *integration tests*

Il gruppo, dopo aver esposto alcuni dubbi sull'attuale implementazione degli *unit tests*, ha chiesto al proponente quale fosse una buona pratica per implementare gli *integration tests*.
Il proponente, dopo aver espresso perplessità sulla necessità di dedicare risorse del gruppo su questo tipo di test, considerati facoltativi da capitolato, è sceso nel dettaglio della domanda andando ad illustrare al gruppo che, a differenza di quanto avviene per gli *unit tests*, l'implementazione degli *integration tests* è molto dispendiosa, richiedendo la definizione del flusso dell'intero test da parte di chi lo implementa.

3.5 Implementazione dei unit tests (divisione service e controller)

Si è discusso sulla corretta implementazione degli unit tests in particolare lato backend. Su nestjs gli unit tests vanno divisi in controller e service. Il primo deve controllare l'esecuzione nei casi eccezionali degli input non validi che però se si esegue la validazione come nella documentazione ufficiale di nestjs (decorator pattern) allora non è necessario; mentre lato service si deve testare se effettivamente la logica della funzione.

3.6 Implementazione del sistema di notifiche

L'incontro è proseguito con una domanda da parte del gruppo riguardo l'implementazione delle notifiche, in quanto il gruppo era a corto di idee in merito agli strumenti che si potrebbero utilizzare.
Il proponente ci ha esposto che, per le necessità del capitolato, quindi quella di non avere notifiche di tipo "*push*", il sistema si potrebbe implementare andando a definire per il "Socket" un nuovo canale di comunicazione apposito per le notifiche, il quale deve trasmettere le nuove notifiche ai clienti già autenticati all'interno del sito, mentre quando un cliente accede al sito, il sito preleverà le notifiche dal database.
Quindi quando nel backend si verifica un evento che dovrebbe generare una notifica, questa dovrà essere salvata nel database e inviare tramite una API al server socket il quale dovrà aggiornare

il client se connesso al socket. Quando il client visualizzerà/gestirà la notifica, questa deve essere eliminata dal database.

3.7 Dubbi sull'architettura da adottare

Di seguito da parte del gruppo è emersa una domanda in merito all'architettura da adottare o che si sta adottando per il progetto. Il fatto che i vari servizi di ordinazione, prenotazione, registrazione sono molto legati tra di loro fa pensare che l'architettura sia più vicina ad un monolite che a microservizi.
Il proponente però ha chiarito che il diagramma esposto non rappresentava un'architettura di tipo monolitico, bensì un'architettura che tende ad essere a microservizi in quanto il fatto che i servizi comunicassero tra di loro tramite chiamate API REST escludeva il monolite. E' stato specificato che la gestione del progetto è una via di mezzo tra le due tipologie, infatti l'architettura può essere associata ad una multi-tier (Architettura multi-tier) dove i vari servizi rappresentano un tier.

3.8 Design pattern utilizzati

Come ultima domanda, il gruppo ha chiesto al proponente quali design pattern fossero da includere all'interno del progetto. Alessandro ha risposto che alcuni sono già implementati, osservando il diagramma esposto, e sono ad esempio:

- *MVC* (Model-view-controller) all'interno del backend;
- *Decorator* all'interno del backend;
- *BFF* (backend for frontend) per quanto riguarda il frontend;
- *Microservizi* per quanto riguarda l'architettura;
- La programmazione orientata agli eventi lato socket.

Il gruppo ha concluso che si impegnerà a documentare tali *design pattern* segnalati.

4 Decisioni prese

- Sistemare lo schema di progettazione secondo le indicazioni del proponente;
- Rivedere la gestione dei campi non validi su Nest.js seguendo la documentazione ufficiale;
- Rivalutare la necessità di implementare *integration tests*;
- Sviluppare il sistema di notifiche tramite socket come suggerito dal proponente;
- Progettare un'architettura di tipo multi-tier;
- Documentare l'utilizzo dei design pattern segnalati.