ISS2025

Fase 1 - Relazione Finale

Bacchelli Valentina
Git repo: https://github.com/VBacchelli/ISS2025/

Introduzione

Durante questa fase del corso, abbiamo sviluppato l'applicazione *ConwayLife*, partendo da una versione base in JavaScript, poi in Java, e aggiungendo gradualmente nuove componenti. Dopo aver implementato una GUI, abbiamo sperimentato diverse tecnologie di comunicazione, fino a trasformare l'app in un microservizio. In questa relazione condivido le mie riflessioni personali sui contenuti trattati e su come abbiano arricchito le mie competenze, sia teoriche che pratiche.

Finalità

Il progetto è stato sviluppato in modo progressivo, partendo da linguaggi e paradigmi familiari (JavaScript, Java e programmazione a oggetti) e ampliandone gradualmente le funzionalità. Inizialmente ci siamo concentrati solo sulla logica di business, interagendo con il software tramite un'interfaccia testuale, il che ha permesso di comprendere meglio ogni parte del sistema prima di integrarle.

L'introduzione di nuove tecnologie è avvenuta gradualmente, in risposta a esigenze specifiche, rendendo il processo più chiaro e motivando ogni scelta. Un aspetto importante è stato il mantenimento di un diario di bordo per tracciare il flusso di pensieri. All'inizio è stato difficile ricordarsi di scrivere o trovare le parole giuste, ma questa pratica aiuta a non dimenticare idee, ottimizzare i tempi e prevenire errori ripetuti. Inoltre, è utile per chiarire il proprio ragionamento, individuare problemi prima ancora di scrivere codice e può essere applicata anche in altri ambiti.

Analisi dei contenuti

Ho sperimentato tutti i sistemi affrontati, anche se con alcune difficoltà. L'introduzione di MQTT è stata la più problematica, inizialmente per errori nel riconoscimento delle librerie (risolti con un clean e refresh del progetto) e poi per problemi di comunicazione. Questo, anche se il messaggio di errore non permetteva di capirlo in modo immediato, si è rivelato un problema piuttosto semplice: è bastato terminare tutti i processi java e poi riprovare.

Non conoscendo MQTT, non ho avuto sufficiente tempo per implementare autonomamente i sistemi che lo utilizzavano e li ho quindi testati solo dopo aver visto il codice già scritto. Sono stata invece in grado di realizzare gli altri sistemi in autonomia.

La scelta di implementare il gioco della vita di Conway non è stata spiegata esplicitamente, ma penso si presti bene ai contenuti trattati. La sua logica semplice è stata un vantaggio, visto che l'obiettivo del corso non era la programmazione complessa, e richiede anche diversi elementi interattivi nella GUI, come pulsanti e celle della griglia. Coinvolge frequenti aggiornamenti di stato da mostrare in tempo reale, spingendo a riflettere sull'ottimizzazione della banda e sulla gestione di più utenti. Tuttavia, anche se ancora non siamo arrivati a implementarlo interamente, credo che il motivo principale per cui è stato scelto questo gioco sia che si presta molto bene ad essere sviluppato a vari livelli di complessità, infatti inizialmente lo abbiamo usato con un'interfaccia testuale senza alcun problema, ma si può espandere fino a trattare ogni cella come un microservizio autonomo. Questo ci ha portati a ragionare sulla scalabilità dell'applicazione e sull'indipendenza dei suoi componenti per poi arrivare ad avere un sistema distribuito.

Abbiamo usato Java per questa implementazione, il che è vantaggioso perché si tratta di un linguaggio già conosciuto da tutti e compatibile con molti framework, come appunto SpringBoot. Inizialmente pensavo che avrebbe avuto più senso usare, ad esempio, React per operare con un virtual DOM che aggiornasse la pagina con solo le nuove modifiche, ma poi ho capito che non sarebbe stato possibile per il fatto che più utenti dovessero visualizzare lo stesso stato di gioco anche connettendosi in seguito. Tuttavia, mi rimane il dubbio che alternative come il framework Express.js o il linguaggio Go possano essere più efficienti e leggeri.

La comunicazione tra il frontend e il backend è stata prima implementata in modalità full duplex e asincrona tramite WebSocket. Questo metodo consente l'interazione machine-to-machine, dove un programma, denominato caller, invia i comandi a ConwayLife25 utilizzando WebSocket per gestire la comunicazione.

Si è deciso poi di passare al protocollo MQTT, ma prima di farlo si è reso il software indipendente dal protocollo di comunicazione utilizzato. In questo modo sarà possibile cambiarlo senza dover modificare le parti esistenti. Questo approccio, noto come *abstraction gap*, consente di separare le funzionalità rendendole più flessibili e modulari.

L'astrazione dal protocollo di comunicazione e dall'implementazione del canale offre diversi vantaggi:

- **Modularità**: il sistema può supportare più protocolli senza dover riscrivere il codice principale.
- **Manutenibilità**: eventuali modifiche o aggiornamenti del protocollo si riflettono solo nelle classi dedicate, senza impattare il resto del software.
- **Scalabilità**: permette di integrare nuovi protocolli o migliorare l'implementazione del canale senza vincoli rigidi.

- **Testabilità**: facilita i test, in quanto il protocollo può essere simulato o sostituito senza modificare l'intero sistema.

Per implementare questo livello di astrazione, sono state sviluppate librerie *custom*, in particolare la *unibo.basicomm23-1.0.jar*. Questa libreria definisce un'interfaccia comune per la comunicazione, permettendo di gestire diversi protocolli senza dipendenze dirette nel codice applicativo. Grazie a questa soluzione, il passaggio a MQTT è stato possibile senza modificare la logica principale del sistema, rendendo il software più modulare e facilmente estendibile.

Il sistema così ottenuto è stato poi suddiviso in modo da rendere completamente indipendenti la logica di business e la GUI. Il risultato presenta alcune caratteristiche tipiche di un'architettura distribuita, in quanto la GUI (*conwayguialone*) e la logica di business e comunicazione (*conway25javamqtt*) sono separati in progetti distinti che interagiscono tra loro. Tuttavia, non sono sicura si possa parlare già di un sistema basato su microservizi; queste sono le mie considerazioni:

- **Indipendenza dei servizi**: Nei microservizi, ogni componente deve essere indipendente e scalabile separatamente. Nel nostro caso, la GUI e la logica di business sono modulari, ma non ritengo siano autonome al punto da essere considerate microservizi veri e propri. Inoltre la gestione della comunicazione è dedicata ai messaggi di questa implementazione specifica e non abbastanza generale a mio parere per essere considerato un sistema a microservizi
- **Comunicazione tramite protocolli leggeri**: L'uso di MQTT come protocollo di comunicazione è coerente con un'architettura a microservizi, dato che permette ai due componenti di interagire in modo disaccoppiato e asincrono.
- **Scalabilità e deployment separato**: Un sistema a microservizi consente il deployment indipendente dei vari servizi. Nel nostro caso, i due progetti sono possono essere eseguiti separatamente e su macchine diverse, e sono anche supportate più istanze in esecuzione (grazie alla decisione di assegnare a un utente il ruolo di owner e ad altri di spettatori), ovvero si ha buona scalabilità.

In sintesi, il sistema sviluppato si avvicina a un'architettura distribuita e introduce principi che potrebbero essere alla base di un'architettura a microservizi. Tuttavia, per definirlo un sistema basato su microservizi in senso stretto, sarebbe necessario un maggiore grado di indipendenza tra i componenti e una gestione più autonoma dei servizi.

Competenze acquisite

Competenze pratiche:

- Inizializzazione di un progetto Gradle da terminale
- Utilizzo di SpringInitializer per creare progetti Gradle
- Utilizzo autonomo di JUnit per l'implementazione di test (visto in precedenza ma senza mai scrivere autonomamente i test)

- Utilizzo di Docker per eseguire e creare immagini
- "Ripasso" dell'utilizzo di SpringBoot, in particolare migliore comprensione delle annotazioni utilizzate nei controller
- Ripasso dell'implementazione della comunicazione tramite WebSocket
- Astrazione dei canali di comunicazione tramite librerie custom, in modo da rendere il sistema indipendente dalla tecnologia usata
- Uso del protocollo MQTT con broker Mosquitto

Competenze teoriche:

- Approfondimento e riflessione sul ruolo del linguaggio (già trattato nel corso di Linguaggi e Modelli Computazionali)
- Differenza tra comunicazione H2M e M2M (human-to-machine e machine-to-machine)
- Cos'è un microservizio (servizio indipendente, di dimensione ridotta, in genere sottoparte di un sistema più grande che comunica con le altre parti senza dipenderne)
- Abstraction-gap, visto in particolare nell'astrazione dal protocollo usato per la comunicazione e nell
- Viste di un sistema software: vista esterna, interna e sommersa
- Concetto di **agente**

Key points

I concetti principali di ingegneria del software che abbiamo potuto richiamare sono:

- **Single Responsibility Principle, Domain Driven Software Development, Divide et Impera**: visti con la rifattorizzazione della business logic fatta nelle prime lezioni, in cui abbiamo creato le classi *Grid* e *Cell* per separare la logica di gioco, che era prima interamente contenuta in *Life*
- **Dependency Inversion Principle**: visto con l'interfaccia IOutDev che fa sì che si possa cambiare il metodo di visualizzazione dello stato di gioco senza cambiare le dipendenze del sistema, basta creare classi che la implementino. Così saranno le altre classi a dipendere dal nostro sistema, e non viceversa, permettendo di non dover toccare la business logic quando cambiano questi dettagli implementativi.
- **Incapsulamento, Abstraction Gap**: si tratta di nascondere i dettagli implementativi all'utente finale. Si è visto sia nell'introduzione della libreria unibocomms per astrarre dall'implementazione del canale di comunicazione usato che nell'incapsulamento della logica di business nel sistema senza dipendenze dirette tra essa e la GUI.

B-