Link GitHub: link

Finalità

In questa prima fase del corso, i contenuti si sono inizialmente concentrati sull'acquisizione di familiarità con alcuni strumenti fondamentali per lo sviluppo. Successivamente, attraverso lo sviluppo incrementale di un'applicazione articolata in varie fasi, ciascuna caratterizzata da livelli di complessità e architetture differenti, l'obiettivo è stato comprendere come anche la creazione di un semplice gioco possa presentare diverse criticità e richiedere approcci architetturali differenti, e come diverse scelte possano influenzare direttamente l'estendibilità e la modularità di un servizio. Si è passati dall'implementazione di un gioco con HTML e JavaScript fino alla realizzazione di due microservizi indipendenti in grado di comunicare tra loro tramite un protocollo specifico.

Le prossime fasi potrebbero concentrarsi sul consolidamento dei concetti appresi nella prima fase, sulla loro estensione e sull'introduzione di nuovi elementi.

Sistemi realizzati

Sono riuscito a realizzare ed eseguire tutti gli esempi svolti a lezione, completandone alcuni a casa. Per motivi di tempo, però, sono riuscito a sviluppare solo una parte delle estensioni assegnate come esercizio.

Abilità/Competenze

Le competenze più pratiche che si possono acquisire in questa fase, riguardano l'uso di strumenti utili per lo sviluppo, come Docker e Gradle. Inoltre, alcune tecnologie, come Spring Boot, possono semplificare il processo di sviluppo, nascondendo i dettagli implementativi di basso livello e mettendo in evidenza quelli di livello superiore.

Per quanto riguarda le competenze più "concettuali", è possibile apprendere che esistono diversi approcci per risolvere un problema e che la scelta del più adatto dipende da molteplici fattori. Le decisioni progettuali influenzano direttamente vari aspetti di un servizio, come l'estendibilità, e l'applicazione dei principi dell'ingegneria del software aiuta a sviluppare sistemi robusti e corretti. Inoltre, la scelta più immediata non è sempre quella migliore.
Un aspetto fondamentale è l'astrazione, che permette di nascondere i dettagli implementativi di basso livello e di mettere in evidenza ciò che è più rilevante.
Sono stati inoltre introdotti i microservizi, un'architettura basata sull'uso di servizi indipendenti che interagiscono tra loro scambiandosi messaggi. È stata sottolineata anche l'importanza della documentazione, scritta sotto forma di diario, che include anche delle riflessioni e considerazioni fatte durante le diverse fasi di progettazione.

Scelta di Conway

La scelta di realizzare il gioco Conway come primo sistema software potrebbe derivare dal fatto che le sue regole non sono troppo complesse e la sua logica è abbastanza semplice da comprendere e realizzare, ma offre comunque la possibilità di creare e sperimentare diversi approcci architetturali.

Scelta di Java

Java non solo è un linguaggio a oggetti che la maggior parte degli studenti ha utilizzato, ma è oggi molto utilizzato grazie alle sue caratteristiche. Perciò la scelta di usare Java è secondo me molto

adatta. Per quanto riguarda Spring Boot, è uno strumento molto diffuso in ambiente Java che semplifica di molto lo sviluppo di applicazioni web o microservizi, come quelli abbiamo realizzato, utilizzando il framwork Spring, pertanto anche questa scelta secondo me è molto adatta.

Keypoints

- Dopo l'analisi dei requisiti e del problema è stato sviluppato il core business senza concentrarsi sulla GUI, in questa prima fase sonon stati utilizzati 3 principi: **Domain Driven Software Developement**, in cui il focus è stato quello di sviluppare del software che rispechiasse il particolare dominio (per esempio le classi e le funzioni sono state nominate in modo che il loro nome evochi la loro funzione), **divide et impera**, in cui il problema viene scomposto in parti più piccole e semplici, e in cui ogni componente si occupa di una parte, e infine **divisione delle responsabilità**, in cui ogni componente dovrebbe avere una singola responsabilità, evitando di accorpare funzionalità non correlate nello stesso elemento.

- Successivamente sono stati sviluppati dei semplici prototipi per vedere se il tutto funzionasse(infatti in una prima versione è la stampa avveniva su console che veniva usato solo come dispositivo di output). Dopodichè, è stato utilizzato Spring per creare una Gui, e ottenere un prodotto più user friendly, basandosi sul principio di **Iron Man**, in cui è stata creata una armatura, rappresentata da Spring, che incapsula il core business già funzionante e testato. L'armatura ha l'obiettivo di aggiungere una Gui modificando pochissime righe di codice. Infatti la relazione tra core business e l'armatura rappresentata da Spring si ha inserendo un controller che si occupa di gestire il gioco (play, stop ecc.), quindi il risultato è che, interagendo con l'armatura, succederà quacosa al suo interno(ovvero al core business). Infatti, per esempio, ora il gioco non va avanti per un numero stabilito di epoche, ma possiamo fermarlo quando vogliamo dicendo "stop" sull'armatura. Il controller è chiamato dal controller di Spring che a sua volta viene attivato dalla Gui(in particolare Spring inietta le dipendeze necessarie, principio di **dipency injection**). La comunicazione avviene tramite WebSocket, in questo modo non bisogna aggiornare la pagina per vedere dei cambiamenti ma il tutto funziona automaticamente grazie alle interazioni asincrone delle WebSocket. Questo approccio apre anche una ulteriore possibilità: utilizzare diversi protocolli sfruttando un meccanismo di astrazione, che consente di concentrarsi solo sul cosa mandare ma non sul come.In aggiunta la Gui è una entità sia di input che di output, infatti è stato possibile creare un player interamente software che comunica con il core business attraverso la WebSocket(una prima interazione M2M). In definitiva abbiamo ottenuto un miscoservizio contenente tutto che abbiamo lanciato anche tramite Docker.

- Dopichè si è passati a 2 microservizi indipendenti che comunicano tramite MQTT: la gui e la parte che implementa il gioco. Anche in questa fase si cerca di rimanere indipendenti dal protocollo utilizzato: principio di **tecnology indipendency**, in cui ci si concentra sulle caratteristiche principali, senza dipendere da una tecnologia specifica. MQTT ha delle dipendenze ben precise (il broker e i topic) ma i 2 microservizi non dovranno necessariamente conoscersi(infatti il gioco parte ma non sa minimamente che esiste la GUI), sarà la libreria MQTT che si occuperà di inviare un messaggio a chi si è iscritto a un particolare topic(il subscriber riceve i messaggi tramite il broker).
  Si cerca di nascondere l'utilizzo di MQTT utilizzando la libreria custom, e infatti non viene esplicitamente eseguita un publish o una subscribe, ma si ha una generica send e receive(concetto di astrazione).
  Vengono anche utilizzati dei messaggi specifici, con ognuno una particolare struttura e semantica.
  In particolare vengono utilizzati 2 topic per far fluire le informazioni dalla Gui a life e viceversa; avremo quindi un topic dove la gui fa publish e life subscribe (*lifein*), su cui la gui manderà le celle iniziali e successivemente il comando di "start", e un topic dove life farà publish e la Gui subscribe (*guiin*) , che viene utilizzato per mostrare sulla Gui l'evoluzione del gioco.
  Infine sono state proposte 2 ottimizzazioni: via WebSocket e per MQTT. Per quanto riguarda

le WebSocket, l'ottimizzazione si basa sull'inviare l'evoluzione delle celle direttamente sulla websocket della Gui, senza passare per il broker, limitando in questo modo il traffico, ma in questo modo si introduce una dipendenza fstidiosa in quanto un dispositivo di livello applicativo deve conoscere l'IP della Gui; viene meno quindi il disaccoppiamento dei 2 microservizi. L'ottimizzazzione per MQTT si basa sull'accumulare per ogni epoca tutti gli update delle varie celle che cambiano stato e inviare un solo messaggio che contiene tutte queste informazioni. Anche qui però qusta ottimizzazione va in conflitto con l'obiettivo di rappresentare ogni cella con un nodo computazionale diverso.

- Infine, si è parlato dei **DSL** (*Domain-Specific Languages*), ovvero linguaggi ad hoc specifici per un particolare dominio o contesto, utilizzati al posto di un linguaggio general-purpose. Un DSL permette infatti di catturare in modo più efficace le caratteristiche di un dominio specifico. In questo contesto è stato introdotto **QAK**.

Sistema software sviluppato

Ritengo che alla fine di questa prima fase il sistema sviluppato sia un sistema basato su microservizi. In particolare, abbiamo 2 microservizi, la gui e la parte di business logic, che rappresentano 2 entità indipendenti che comunicano scambiandosi informazioni; nello specifico la comunicazione avviene sfruttando il protocollo MQTT basato su un'architettura publish/subscriber.

Librerie custom

Le librerie custom, secondo me, hanno il ruolo principale di fornire delle soluzioni apposite che rispecchiano le specifiche del problema, inolte, permettono sia di dividere meglio il codice, sia di riutilizzarlo in futuro o in diverse parti dello stesso progetto.
Le classi relative al protocollo MQTT sono state sviluppate per gestire la comunicazione tramite questo protocollo, tenendo conto del principio di astrazione; in fase di inizializzazione si inserice solo il nome, l'indirizzo del broker e i topic di input e output, sarà poi la libreria a occuparsi di gestire il tutto. Inoltre, i messaggi che vengono scambiati possono essere delle semplici stringhe, oppure dei particolari tipi di messaggi con una diversa semantica.

Ruolo dei linguaggi di Programmazione

I linguaggi di programmazione permettono di scrivere dei programmi utilizzando un linguaggio e delle strutture di più alto livello. Sfruttando le regole e i principi legati a un linguaggio è possibile applicare diversi pattern di progettazione e, infine, ogni linguaggio ha caratteristiche diverse che permettono di scegliere quello più adatto in base al contesto.