UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Magistrale in Informatica

Advanced Tools and Techniques for Software Development

REHEARSAL ROOM SCHEDULE WEB SERVICE

FRANCESCO TERROSI

6326113

Anno Accademico 2018-2019

INDICE

1

INTRODUZIONE AL SOFTWARE

1.1 DESCRIZIONE E VINCOLI

Il software sviluppato implementa un web service per la gestione delle prenotazioni di una sala prove.
Lo sviluppo del software è iniziato definendo dei requisiti sull'orario della sala e sulla definizione delle richieste di prenotazione valide. In particolare:

- La sala prove in questione offre 3 sale, prenotabili per turni di 2 ore e 30 minuti
- Ciascuna sala è prenotabile per qualunque orario (il minutaggio non deve necessariamente essere un multiplo di 30, la sala prove rimane aperta h24)
- È possibile effettuare una prenotazione fino a 5 minuti prima dell'orario specificato
- Tutte le prenotazioni richiedenti un orario O, $O \leqslant \texttt{ora attuale} + 5\ \texttt{minuti}$ vengono rifiutate
- Soltanto gli utenti registrati nel sistema possono effettuare le prenotazioni
- Le prenotazioni vengono associate in maniera univoca all'username scelto in fase di registrazione
- Non possono esistere due utenti con lo stesso username

1.2 PANORAMICA DEL SOFTWARE

Il software è stato implementato utilizzando diverse tecniche di programmazione fra cui il *Test-Driven-Development*, *Build-Automation* (*Maven* e

Gradle), *Continuous Integration*... seguendo una filosofia di sviluppo modulare. Possiamo dividere i vari pacchetti in pacchetti di **utilità** (model, exceptions, configurations...) e pacchetti di **servizio** (services, repository, web)

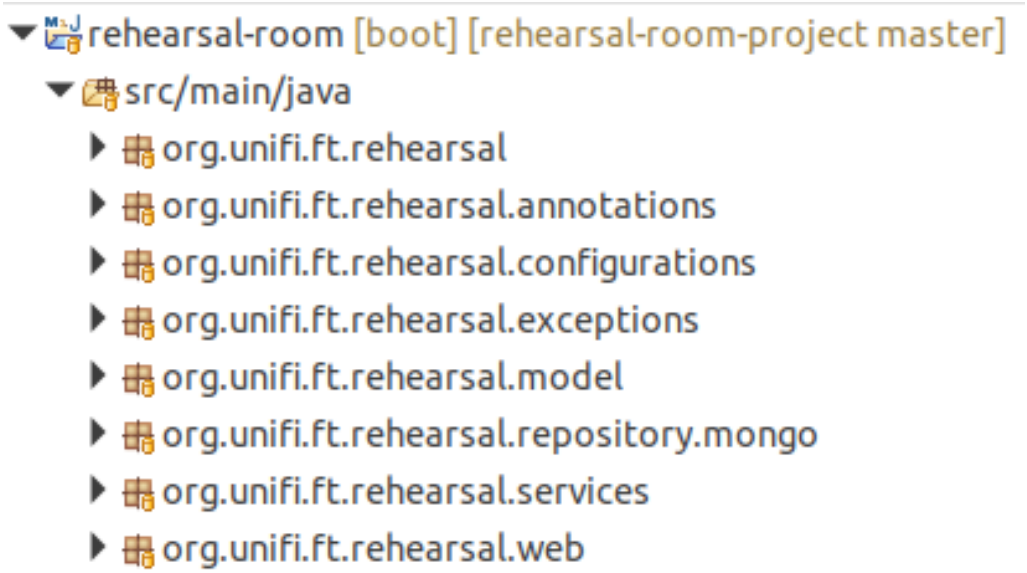

Figura 1: Gerarchia dei pacchetti

L'applicazione è stata sviluppata utilizzando il *Framework Spring Boot*, grazie al quale è stata facilitata l'implementazione e la gestione di un database MongoDB (che consiste di due differenti repository: uno per gli utenti e uno per le prenotazioni) e l'esposizione dei metodi offerti dai vari *services* grazie ai Web Controller.

Le operazioni sui repository sono implementati appunto dalle classi contenute in `org.unifi.ft.rehearsal.services`: BandService e Scheduler.
BandService si occupa della gestione della registrazione nel sistema dei gruppi che vogliono usufruire della sala prove. Le operazioni possibili sono il salvataggio di un utente nel sistema e la ricerca di un determinato utente tramite *username*.
La classe Scheduler implementa invece le operazioni di salvataggio delle prenotazioni. Essendo questo servizio utilizzato dagli utenti sono fornite diverse operazioni per il salvataggio, la ricerca e la cancellazione.

Come accennato all'inizio del paragrafo, questi servizi sono esposti tramite un'interfaccia web realizzata grazie al *framework Model View Controller* offerto da *Spring Boot*, di cui verrà approfondito il funzionamento nei

paragrafi successivi.

2

TECNICHE E FRAMEWORK UTILIZZATI

2.1 VERSION CONTROL SYSTEM

Per controllare il versionamento del software durante tutta la fase di sviluppo è stato utilizzato il software *git*, in congiunzione con la piattaforma *github*:

Maven - https://github.com/FrancescoTerrosi/rehearsal-room-project

Gradle - https://github.com/FrancescoTerrosi/rehearsal-room-gradle

Nonostante il lavoro sia stato svolto da un singolo studente è stato comunque adottato il modello *gitflow*, accompagnato dai meccanismi di pull-request offerti da *github*, grazie ai quali è stato possibile effettuare vari check di integrità della build su *travis*, *coveralls* e *sonarcloud*.

2.2 BUILD AUTOMATION

Come già detto, sono stati usati due strumenti di build automation:

- Maven: solido, conosciuto e ben accettato nell'ambiente di sviluppo software, data la sua diffusione vi sono molti tutorial, plugin e schemi di configurazione
- Gradle: molto recente, in continuo sviluppo, permette una configurazione pressoché totale della build del progetto

2.2.1 *Maven*

Le specifiche di un progetto Maven vengono definite nel file "pom.xml". All'interno di questo file è possibile definire alcune opzioni di configurazione del progetto (nome del gruppo e del progetto, versione di Java...) e,

soprattutto, definire le dipendenze necessarie al corretto funzionamento del software e i plugin che specificano le operazioni da fare durante il processo di build.
Per adottare una sorta di approccio modulare anche nel processo di build del progetto, sono stati definiti diversi profili all'interno del pom.
Definire un profilo permette di incapsulare i plugin e le loro configurazioni al suo interno, in modo tale che queste vengano attaccate alla fase appropriata del lifecycle di Maven solamente quando richiesto.
Per separare le operazione di generazione dei report di *JaCoCo*, i *Mutation Tests*, gli *Integration Tests* e gli *End to End Tests* sono stati quindi definiti i profili:

- jacoco
- mutation
- integration
- e2e

```
mvn clean verify -Pjacoco coveralls:report sonar:sonar
mvn verify -Pintegration
mvn verify -Pe2e
```

Figura 2: Fasi e profili utilizzati per effettuare la build del progetto su travis-ci. Il profilo *mutation* non viene utilizzato in quanto di scarsa rilevanza per i check successivi (ma molto utile in locale)

2.2.2 *Gradle*

Gradle è, attualmente, il maggior *competitor* di Maven per quanto riguarda la build automation.
La procedura di assemblamento di un software è intrinsecamente diversa: se in Maven abbiamo un ciclo di vita rigido e ben definito, in cui i vari *goal* vengono eseguiti in un ordine ben definito e in maniera sequenziale, Gradle si basa sulla definizione di *tasks* (intuitivamente possiamo vederli come un corrispetivo dei *goal*), il cui ordine di esecuzione viene definito dal *task graph*, un grafo aciclico che definisce l'ordine di esecuzione dei vari compiti.

Il file in cui vengono definite le dipendenze del progetto, i plugin e

altre configurazioni è il file "build.gradle". Uno dei vantaggi è indubbiamente l'utilizzo di un DSL *Groovy-like,* al posto del verboso xml utilizzato da Maven.

```
// sezione in cui vengono definiti i binary plugins
// importano classi che eseguono operazioni necessario alla build
plugins {
    id 'org.springframework.boot' version '2.1.3.RELEASE'
    id 'java'
    id 'jacoco'
    id 'com.github.kt3k.coveralls' version '2.8.2'
    id "org.sonarqube" version "2.7"
    id 'com.sourcemuse.mongo' version '1.0.7'
    id 'info.solidsoft.pitest' version '1.4.0'
}

// applicazione degli script plugin
// implementano configurazioni aggiuntive alla build gradle
apply from: "$rootDir/itest.gradle"
apply from: "$rootDir/e2etest.gradle"
apply plugin: 'io.spring.dependency-management'

// sezione delle dipendenze del progetto
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-web'

    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.springframework.security:spring-security-test'
}
```

Figura 3: Esempio di file build.gradle

Gradle permette inoltre di definire dei *task* personalizzati che possono essere inseriti all'interno del task graph (e quindi eseguiti ogni volta che viene eseguita la build del progetto. In questo caso possiamo immaginarli, con un certo livello di astrazione, come un *goal* di Maven che viene attaccato a una fase) oppure eseguirli in maniera autonoma.

Uno degli aspetti fondamentali nella definizione di nuovi task Gradle è la loro indipendenza. È buona pratica definire un file .gradle apposito in cui vengono dichiarati i *source sets* e le *dipendenze* necessarie al compimento del task, di modo tale che possa essere eseguito indipendentemente dagli altri. È comunque possibile definire un ordinamento fra essi, utilizzando i comandi *dependsOn* e *finalizedBy*.

```
// Definisce le cartelle sorgenti
sourceSets {
    integTest {
        java.srcDir 'src/it/java'
        resources.srcDir file('src/test/resources')
        compileClasspath += sourceSets.main.runtimeClasspath
        runtimeClasspath += output + compileClasspath + configurations.testRuntime
    }
}

// Definisce le dipendenze del task integTest. È possibile ereditare
// le dipendenze di 'testCompile' ma in tal caso avremmo dovuto
// manualmente eseguire il task test prima di ogni esecuzione di integTest
dependencies {
    integTestCompile 'org.springframework.boot:spring-boot-starter-test'
    integTestCompile 'org.springframework.security:spring-security-test'
}

// Dichiarazione del task 'integTest'
task integTest(type: Test) {
    description = 'Runs the integration tests.'
    maxHeapSize = '1024m'

    testClassesDirs = sourceSets.integTest.output.classesDirs
    classpath = sourceSets.integTest.runtimeClasspath

    mustRunAfter tasks.test
    check.dependsOn integTest
}
```

Figura 4: Esempio di task definito dall'utente nel file itest.gradle

Un altro degli aspetti fondamentali di Gradle è il massiccio utilizzo del Gradle Wrapper.
Il wrapper consente di poter effettuare build gradle senza averlo installato sulla propria macchina. Lavorando in team di sviluppo il vantaggio è

ancora maggiore in quanto assicura che tutti i membri effettuino le build utilizzando la stessa versione di Gradle, senza bisogno di alcuna gestione aggiuntiva.

Tutti i plugin e le dipendenze Maven del progetto sono state ritradotte in Gradle, per assicurare l'uniformità delle due versioni. Su travis, la build viene eseguita con il seguente comando:

```
./gradlew build jacocoTestReport coveralls sonarqube
```

Figura 5: Viene invocato il gradle wrapper per inizializzare la versione corretta di Gradle con cui eseguire la build del progetto

I task che vengono eseguiti sono:

- build
  - → Configura la build e calcola il task graph, compila i file sorgente andando a scaricare le dipendenze, esegue i test e produce il fatjar dell'applicazione
- jacocoTestReport
  - → Esegue JaCoCo per calcolare la code coverage
- coveralls
  - → Cerca il file .xml prodotto da JaCoCo e invia i risultati a coveralls
- sonarqube
  - → Effettua l'analisi su SonarCloud

È possibile apprezzare come le operazioni per assemblare e testare effettivamente il progetto siano tutte racchiuse nel comando (task) *build*.

2.3 CONTINUOUS INTEGRATION

Per entrambi i progetti è stata stabilita un link con i server di Travis-CI, in modo tale da poter eseguire le build complete in remoto e poter continuare a lavorare localmente sul software.
Le specifiche della build sono visualizzabili nel file .travis.yml, un file di configurazione che serve per dare direttive alla virtual machine su cui il progetto viene assemblato.

Fra le due versioni vi è una piccola differenza: nel progetto Maven, i server MongoDB sono stati simulati attraverso l'utilizzo di Docker; per poterlo utilizzare sulla piattaforma di Travis è necessario specificarlo fra i servizi desiderati in modo tale che esso possa venire scaricato e successivamente eseguito dal plugin fabric8 in fase di testing.
Nel progetto Gradle è stato invece utilizzato un plugin per avere un server MongoDB *embedded* nell'applicazione, pertanto non è stato necessario specificare l'utilizzo di Docker nel file .travis.

2.4 FRAMEWORKS

L'applicazione è stata sviluppata interamente utilizzando il framework *Spring Boot*, una delle varie soluzioni proposte dallo *Spring Framework* per la creazione di web-application in Java.
L'utilizzo del framework in questione oltre ad utilizzare il meccanismo di inversione del controllo proprio di Spring, permette di sviluppare applicazioni web in maniera molto rapida, rispetto all'approccio tradizionale basato su servlet.

Uno degli aspetti più interessanti è appunto l'implementazione del meccanismo di *inversion of control* tramite la *dependency injection*. Dal momento che le applicazioni sviluppate con Spring Boot vengono eseguite in un web-container dove, al suo interno, viene a sua volta eseguito un server (Tomcat, in questo caso), utilizzando il meccanismo dei server embedded proprio di Spring.

Dal momento che risulta pressoché impossibile avere un controllo totale sull'istanziazione degli oggetti una volta che l'applicazione viene lanciata, il meccanismo di *dependency injection* è di estrema importanza. Nelle applicazioni Spring Boot, al momento del loro avvio, viene caricato un *application-context* in cui sono contenute tutte le informazioni relative ai *Bean* che non sono stati inizializzati. Attraverso l'utilizzo di questo *contesto* è possibile dunque iniettare i *Bean* nei vari oggetti al momento dell'inizializzazione dell'applicazione. Spesso è necessario definire dei file di *configurazione* per fornire informazioni aggiuntive sui Bean da creare.

Un'altra importante *feature* offerta da Spring Boot sono le API per i server MongoDB. Attraverso la specifica di un semplice Bean (mongoTemplate) e la definizione dell'indirizzo IP e la porta su cui esso deve girare, è possibile connettersi al server e utilizzarlo in maniera molto

intuitiva, grazie anche all'interfaccia MongoRepository.
Altre feature degne di nota:

- Thymeleaf: Template Engine utile per il rendering delle pagine HTML
- Spring-Security: Permette di implementare un meccanismo di autenticazione basato su Username e Password

3

CONCLUSIONI

Lo scopo di questo progetto era quello di condurre un'analisi più approfondita sulla possibilità di riconoscere gli utenti e le frasi da loro pronunciate durante una conversazione Skype e di riuscire a distinguere fra videochiamate, conferenze e chiamate fra due utenti.
Purtroppo i dati riportati ci dimostrano come alcuni di questi obiettivi siano impossibili da raggiungere, in particolare non è stato possibile:

1. Identificare elementi esterni alla conversazione
2. Identificare tratti distintivi nella parlata di un utente
3. Stabilire la lingua della conversazione (a meno di utilizzo di software come Skypegrep)

L'impossibilità di questi 3 punti (salvo casi specifici per il punto (1)) è da attribuire quasi completamente al codec a bitrate variabile utilizzato da Skype per la cattura dei messaggi. In questo modo è impossibile riconoscere uno specifico utente, nè è possibile catturare rumori esterni. Rimane tuttavia possibile identificare *specifiche* frasi all'interno di una conversazione.

In ogni caso sono stati ottenuti risultati interessanti per quanto riguarda:

1. La possibilità di distinguere se è in corso una chiamata, una videochiamata o una conferenza
2. La possibilità di capire se vi sono fonti di rumore *costante* esterno alla conversazione
3. A seguito di un adeguato train-set, la possibilità nella maggior parte dei casi identificare le frasi o le parole pronunciate

Per quanto riguarda il terzo punto è importante ribadire che non è esattamente la frase ad essere riconosciuta (ovvero non è possibile, analizzando i pacchetti, capire *quale* frase sia stata pronunciata) ma è possibile osservare la presenza di determinate sequenze di pacchetti, riconducibili a frasi specifiche.

Nel condurre gli esperimenti, come già è stato detto all'inizio di questo documento, sono stati applicati dei filtri con *Wireshark* in modo tale da ridurre al minimo le fonti di incertezza sui dati. È utile notare tuttavia che:

- È possibile che un'elevata latenza di rete disturbi la qualità delle osservazioni
- Nel monitoraggio delle conferenze non è sempre possibile stabilire il destinatario dei pacchetti, costringendo ad applicare meno filtri e quindi a catturare anche pacchetti non relativi al traffico Skype
- Nell'analisi delle videochiamate e delle conferenze sono stati stabiliti dei bound arbitrari per la cattura dei pacchetti audio. Per quanto i risultati fossero in linea con le precedenti osservazioni non è da escludere che alcuni pacchetti siano stato involontariamente esclusi