Briscola Online - Report Finale

Progetto in *Sistemi Distribuiti*

Andrea Zammarchi
andrea.zammarchi3@studio.unibo.it

March 2023

Indice

1 Abstract

Il progetto prevede la realizzazione di una versione online multiplayer del famoso gioco di carte Briscola.

Le partite saranno esclusivamente composte da due giocatori, in modalità 1vs1. Le carte utilizzate per giocare a Briscola sono 40, raggruppate in 4 gruppi definiti "semi" (coppe, denari, bastoni, spade). La tipologia di carte sarà quella Romagnola. I punti totali presenti nel mazzo sono 120; per decretare il vincitore è necessario raggiungere un minimo di 61 punti, mentre se i punti raggiunti sono 60, si verifica una situazione di parità.

2 Obiettivi

2.1 Q/A

- *Cosa succede una volta avviato il gioco?* → L'utente seleziona il suo nickname personale (non già utilizzato da un altro utente online) ed effettua l'accesso.
- *Come è possibile partecipare a una partita?* → Una volta eseguito il login, l'utente potrà scegliere di unirsi ad una lobby (creata da un altro giocatore) oppure crearne una nuova lui. Quando una lobby viene riempita da esattamente due giocatori è possibile iniziare la partita.
- *Come si svolge una partita?* → Tramite un'interfaccia grafica l'utente può decidere di giocare una delle carte che ha in mano quando è il suo turno. Quando entrambi i player hanno giocato una carta, la presa va a quello che ha giocato la carta con peso maggiore.
  - Il valore delle carte va nel seguente ordine: 2-4-5-6-7-fante-cavallo-re-3-asso.
  - Il seme di briscola è quello corrispondente dell'ultima carta del mazzo, che viene mostrata fin dall'inizio del match.
  - Se un utente gioca una carta col seme di briscola, questa ha peso maggiore su tutti gli altri semi.
  - Se vengono giocate due carte con semi diversi (e nessuno dei due è il seme di briscola), ha peso maggiore la carta giocata per prima.

  Dopo che un giocatore ha conquistato una presa pesca un altra carta dal mazzo, dopodichè fa lo stesso l'avversario. Quando le carte nel mazzo sono finite e i due avversari hanno giocato tutte le carte nelle loro mani, la partita è terminata. A fine partita si conta il valore di tutte le prese fatte da ogni giocatore, colui che fa almeno 61 punti vince. Se entrambi totalizzano 60 punti, finisce in parità. Il valore delle carte è il seguente:
  - 2-4-5-6-7 → 0 punti
  - fante → 2 punti
  - cavallo → 3 punti
  - re → 4 punti
  - 3 → 10 punti
  - asso → 11 punti
- *Cosa succede a fine partita?* → Una volta decretato il vincitore, i giocatori vengono riportati alla finestra della lobby dove possono decidere se ri-sfidarsi o abbandonare la lobby. Un utente può decidere in qualsiasi momento di abbandonare una partita.

2.2 Scenari

Di seguito viene riportato il Diagramma dei Casi d'Uso (Figura 1) che rappresenta tutti gli scenari di Briscola Online dal punto di vista dell'utente.

In particolare abbaimo l'attore base (*Utente*) che può disconnettersi dal gioco in qualsiasi momento. Tutti gli altri attori raffigurano sempre l'utente in momenti diversi:

- All'avvio dell'applicazione l'utente è un *Utente non registrato* e può semplicemente inserire un nickname.
- Appena inserito il nickname, l'attore principale è un *Utente registrato* che può creare una nuova lobby o unirsi ad una esistente.
- Una volta dentro una lobby, l'*Utente in lobby* decide se avviare la partita (solo se la lobby è completa) oppure abbandonarla.
- Avviata la partita l'*Utente in partita* può o giocare una carta (se è il suo turno) altrimenti abbandonare la partita. Terminata la partita, l'attore principale torna a essere l'*Utente in lobby*.

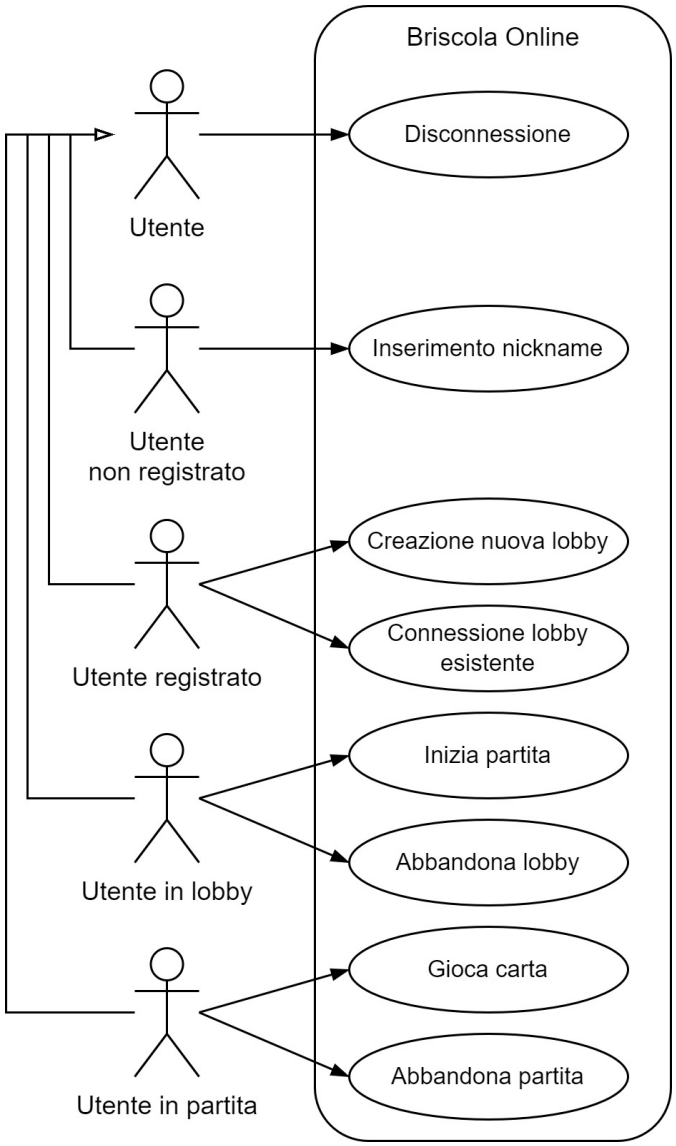

Figura 1: Diagramma dei Casi d'Uso

2.3 Politiche di autovalidazione

Il gioco deve poter girare su qualsiasi client dotato di una JVM con una versione di Java maggiore/uguale alla 17.

L'elenco delle lobby aperte/chiuse deve essere aggiornato frequentemente. Inoltre, devono essere costantemente sincronizzati i client dei giocatori all'interno della stessa lobby, in modo da garantire la coerenza del gioco (ad esempio prese, carte giocate, punteggi, ecc).

Il software prodotto ha come obiettivi principali una buona *scalabilità*, una discreta *fluidità* del gioco e un'adeguata *Fault Tolerance*. In particolare, il Web Service deve essere in grado di seguire un elevato numero di client contemporaneamente, il gioco deve essere eseguito in maniera fluida sul client (senza problemi di performance grafiche e non grafiche), infine non deve bloccarsi in caso di errori.

Il collaudo dell'intero sistema viene effettuato in maniera automatizzata tramite test unitari e di integrazionie per mezzo del framework JUnit. Esso viene verificato nei suoi vari livelli di astrazione: dai test base di connettività ai test di logica di un'intera partita.

3 Analisi dei Requisiti

Requisiti impliciti Per il corretto funzionamento dell'applicazione è necessario che ogni client esegua correttamente richieste al server principale. Il tipo di approccio scelto è quello dove il server si occupa della gestione delle lobby e della logica delle varie partite fra due client. In questo modo vengono limitati i compiti logici al client, permettendo di ridurre al minimo la possibiltà di incotrare lag grafici da parte degli utenti.

Requisiti non funzionali Il gioco deve anche puntare ad un'alta facilità d'uso per gli utenti, proprio per questo motivo si è deciso di aggiungere anche un'interfaccia grafica (acnhe se minimale, in quanto non è l'obiettivo di questo corso creare una GUI avanzata). L'interazione deve essere soprattutto spontanea e intuitiva, riducendo al minimo il numero di click necessari per interagirci.

Paradigmi e tecnologie: usati VS ideali Al fine di soddisfare i requisiti evidenziati, si è optato per un'architettura Client/Server, dove il server in particolare viene implementato come Web Service, dotato di ReST API e che utilizza HTTP come protocollo di comunicazione.

Per cercar di avvicinarsi il più possibile all'obiettivo di avere un'elevata scalabilità del sistema, si è deciso di implementare il Web Service tramite l'ausilio del framework Java `Javalin` [6], in quanto semplice (richiede poche linee di codice), prestazionale (ottima scalabilità) e flessibile (supporta la programmazione asincrona). Esso infatti è un estensione di `Jetty` [7], uno tra i più utilizzati e stabili web-servers sulla JVM.

4 Design

Di seguito si andranno ad elencare e motivare le scelte di design adottate.

4.1 Struttura

Nella figura 2 viene mostrato il Diagramma delle Classi per quanto riguarda le entità principali del Model. Per evitare di appesantire eccessivamente lo schema, non sono stati riportati i *getter/setter* dei vari attributi.

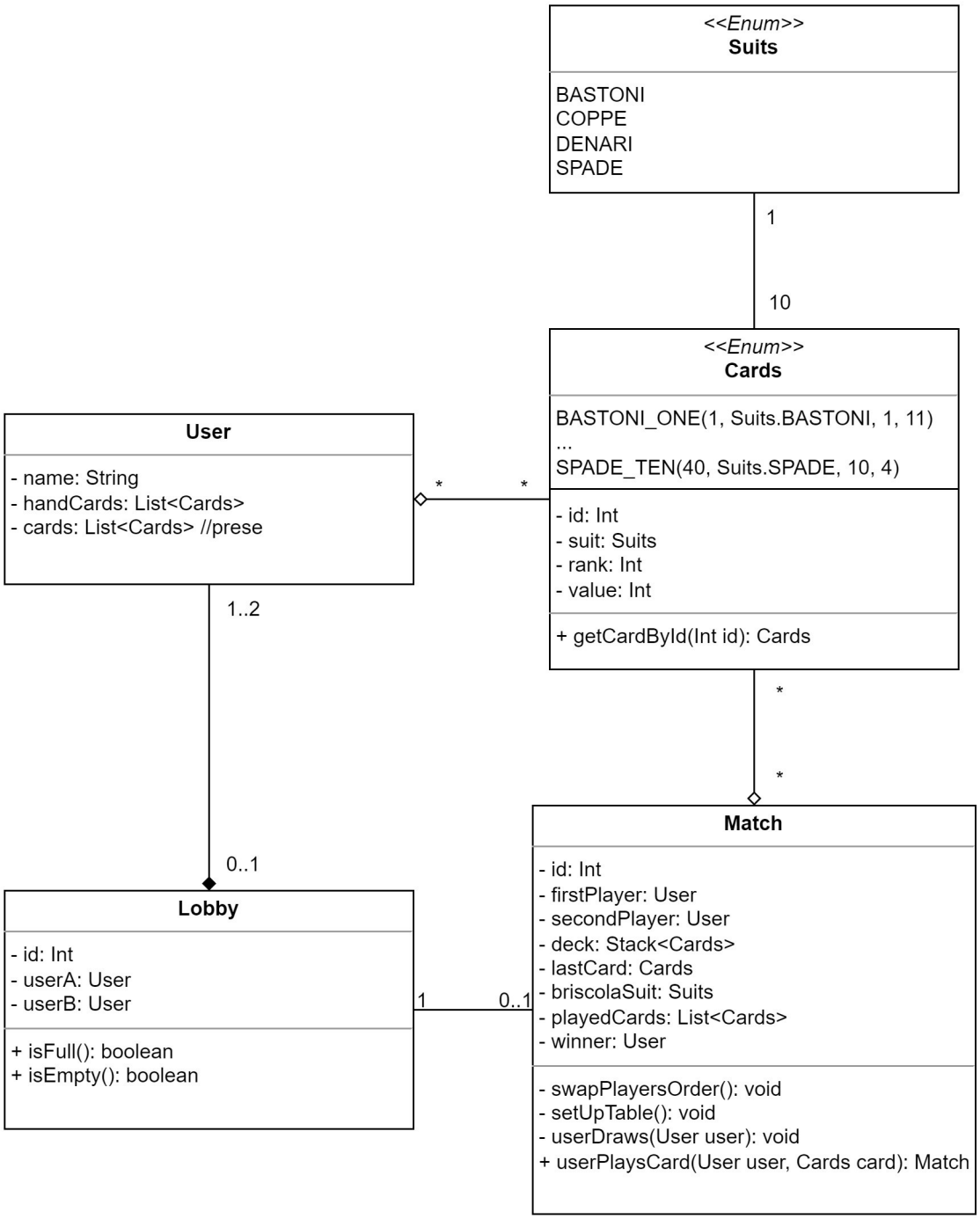

Figura 2: Diagramma delle Classi del Model.

Suits L'entità rappresenta semplicemente il concetto di *seme* di una carta. È implementata come una `Enum`, contenente i quattro semi: *bastoni*, *coppe*, *denari*, *spade*.

Cards L'entità rappresenta il concetto di carta, la quale possiede:

- un *id* identificativo;
- un *seme* di appartenenza;
- un *rank* che indica il suo peso (utilizzato per calcolare la carta con peso maggiore);

- un *value* che indica il suo valore (utilizzato al momento del conteggio dei punti totali delle prese).

È implementata come una `Enum`, contenente tutte le 40 carte da gioco.

User L'entità rappresenta il concetto di un normale utente di gioco, il quale possiede:

- un *name* identificativo;
- una lista di carte che ha in mano durante una partita (*handCards*);
- una lista di carte che è l'insieme delle prese che ottiene durante una partita.

Lobby L'entità rappresenta il concetto, appunto, di una *lobby* di gioco, possiede:

- un *id* identificativo;
- almeno un utente (*userA*/*userB*) e al massimo due.

Match L'entità rappresenta il concetto di partita, contenente:

- l'*id* identificativo della lobby dalla quale è stata creata;
- i due utenti *firstPlayer* e *secondPlayer* (*first* e *second* indicano il turno durante il gioco, quindi l'utente *firstPlayer* sarà il primo a giocare una carta);
- uno `Stack` di carte, *deck*, ovvero il mazzo;
- *lastCard* è l'ultima carta del mazzo e rappresenta il seme di briscola *briscolaSuits*;
- *playedCards* è la lista delle carte giocate dai due utenti;
- *winner* è l'utente che esce vincitore alla fine della partita.

Questa entità contiene anche alcuni metodi fondamentali per la logica di gioco, molti dei quali privati per incapsularne il funzionamento. Tra i più importanti vi sono *swapPlayersOrder* che inverte il turno dei giocatori, *setUpTable* "distribuisce" le carte sul tavolo, *userDraws* dove l'utente indicato pesca una carta dal mazzo (se ancora non è finito). L'unico metodo pubblico è *userPlaysCards*, esso è stato reso pubblico in quanto viene richiamato dal client esterno verso il server quando un utente decide la carta da giocare.

4.2 Comportamento

In Figura 3 viene mostrato il Diagramma degli Stati dalla prospettiva di un utente, dal momento in cui avvia l'applicazione al momento in cui viene avviato un match. L'applicazione può terminare in qualsiasi momento nel momento in cui l'utente decide di chiudere la stessa.

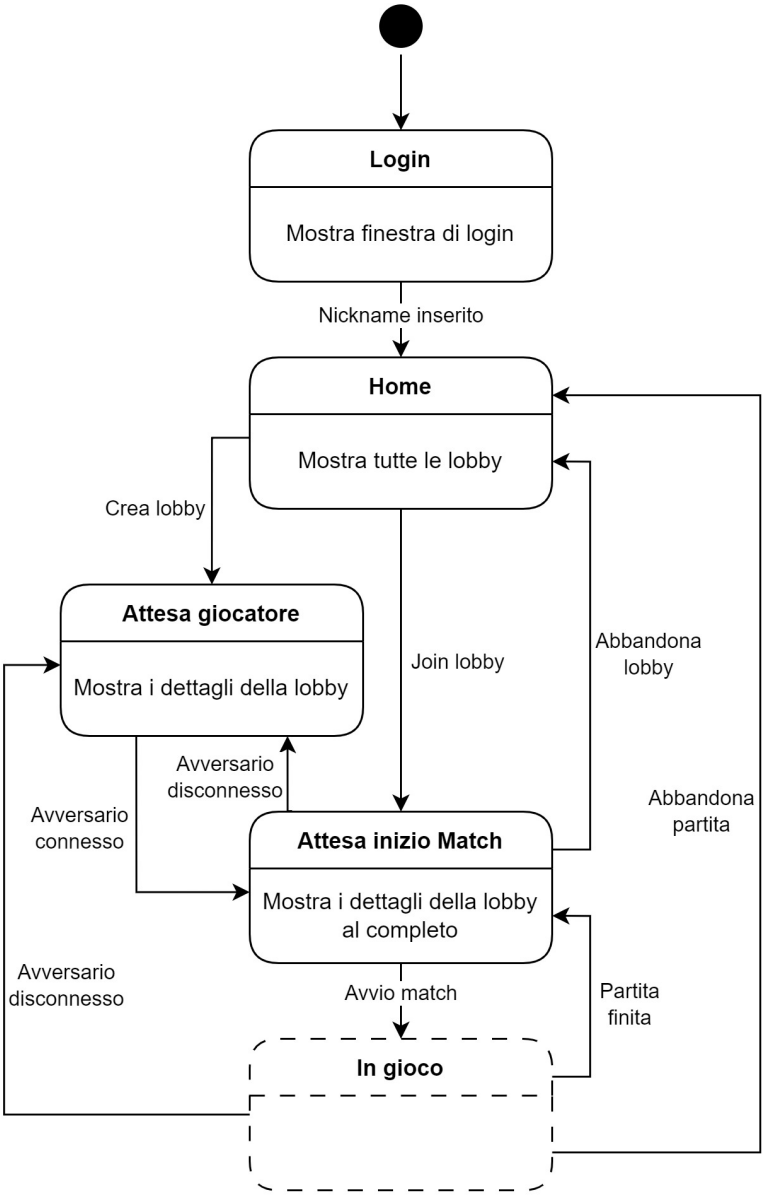

Figura 3: Diagramma degli Stati - Connessione lobby.

Inizialmente l'utente avvia l'applicazione e inserisce un nickname non già in uso. A questo punto viene mostrato il *Menu Home* dove è visibile l'elenco di tutte le lobby e selezionandone una si unirà a questa. Altrimenti ne crea una nuova. In ogni caso verrà mostrato il *Lobby Menu* con i dettagli della lobby (che può essere piena oppure no).

Quando una lobby diventa al completo, un qualsiasi utente può avviare la partita. In Figura 4 viene mostrato il Diagramma del Stati della logica di un match.

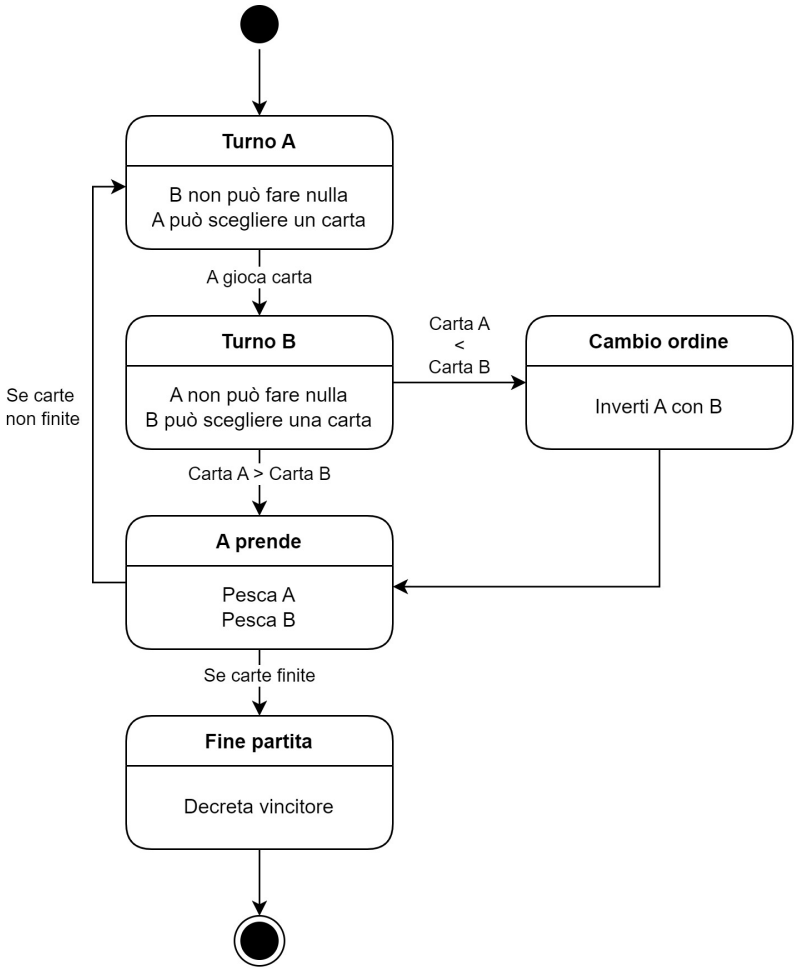

Figura 4: Diagramma degli Stati - Logica Match.

La logica di una partita è la stessa del gioco fisico ufficiale di Briscola [1] e segue le stesse regole. In caso di improvvise o volute disconnessioni, il server gestisce perfettamente tale evento ed agisce di conseguenza in base alla situazione, ovvero, a seconda se l'utente era in un match o meno.

4.3 Interazione

Per scambiare informazioni tra client e server si è deciso di scrivere i messaggi utilizzando il formato Json, attraverso la serializzazione/deserializzazione di oggetti, con l'ausilio della liberia `Gson` [4].

Nella Figura 5 viene mostrato il Diagramma di Sequenza per quanto riguarda una tipica sequenza di scambio di informazioni tramite richieste HTTP tra client e server. Viene preso come esempio la creazione di una nuova lobby da parte di un client A e la connessione di un client B alla lobby appena creata.

Per facilitare la comprensione dello schema è necessario sottolineare che, a livello di codice, è stata creata un'interfaccia **Briscola** che viene implementata in due classi distinte: **LocalBriscola** lato server (dove vengono aggiornate tutte le entità attive del sistema, ovvero User, Lobby e Match) e **RemoteBriscola** lato client (dove a ogni metodo corrisponde una richiesta HTTP da inviare al server, la quale richiamerà un metodo all'interno di LocalBriscola).

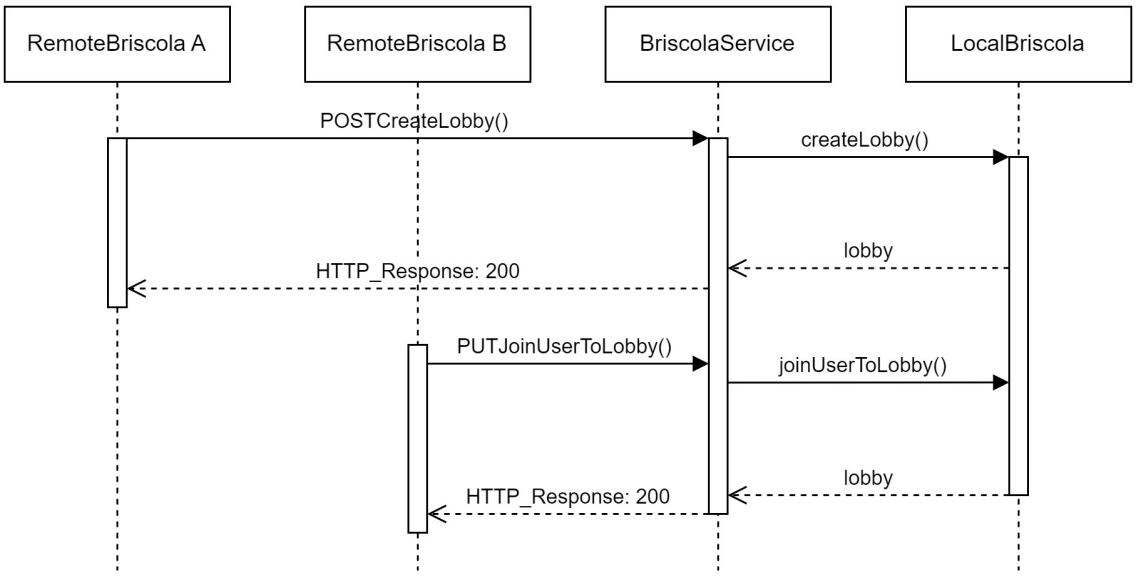

Figura 5: Diagramma di Sequenza - Connessione lobby.

La sequenza di azioni è analoga per ogni altro tipo di operazione che coinvolga lo scambio di informazioni tra client e server, anche per le entità User e Match, e per evitare ripetitività non vengono riportate.

5 Dettagli Implementativi

In questa sezione verranno riassunti i meccanismi e le strategie adottate nella fase di implementazione, citando brevemente le tecnologie utilizzate. L'intero sistema consiste in un progetto Java multi-modulo. In particolare è suddiviso in quattro moduli:

- `common`
- `client`
- `server`
- `test`

5.1 Common

Il modulo `common` contiene tutti gli elementi del *Model* (Suits, Cards, User, Lobby, Match) comuni al client e al server.

Inoltre, implementa anche la parte di *presentation*, ovvero la serializzazione/deserializzazione in JSON degli oggetti Java, tramite il framework *Gson*[4].

5.2 Client

L'implementazione del `client` segue il pattern *MVC* (Model View Controller). In particolare, il package *view* fornisce un'interfaccia grafica (sfruttando il framework *JavaFX*[5]) con la quale l'utente può interagire. Il Model, come già detto, è contenuto nel modulo `common`. Il package *controller*, invece, invia e riceve costantemente aggiornamenti del Model dal server tramite l'invio di richieste HTTP e applica le modifiche alla *view*.

Le richieste HTTP verso il server fanno principalmente riferimento ai seguenti URL:

- */users*
- */lobbies*

- */matches*

Tali richieste possono essere di tipo GET, POST, PUT o DELETE, con l'aggiunta (se necessario) di parametri.

Per l'elenco dettagliato di tutte le possili richieste vedi paragrafo successivo.

5.3 Server

Il server è stato implementando tramite l'ausilio di `Javalin`[6]. Nel particolare si tratta di un **ReST-full web-service**, dotato di **Api**. Esso agisce su tre *resources* principali: *User*, *Lobby* e *Match*. Il funzionamento del sistema in generale può essere chiarito dalla Figura 6

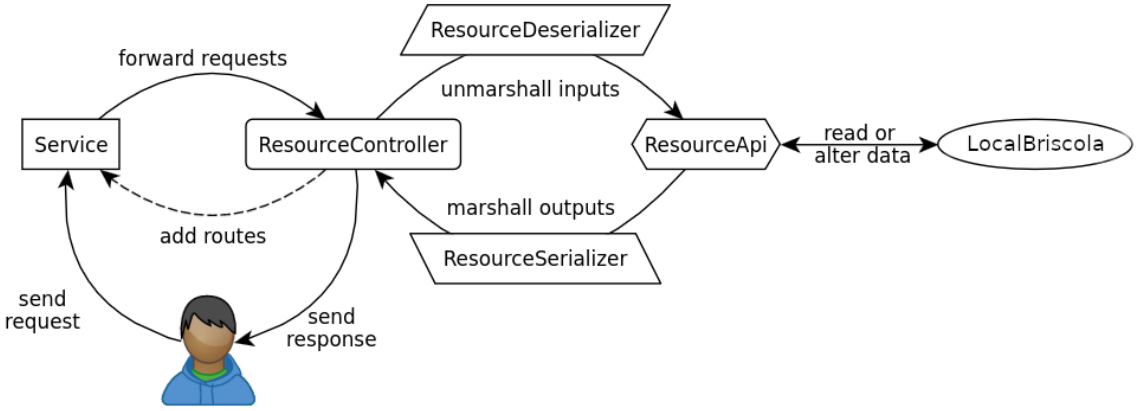

Figura 6: Struttura del progetto.

Per ogni *resource* è associato un *resourceController* che riceve le richieste dai client e le indirizza alla *resourceApi* corrispondente. Quest'ultima agisce su un istanza locale del database del gioco (*localBriscola*), implementato come `singleton`.

Tutte le rotte registrate nel web server sono state documentate con l'ausilio di **Swagger**[9] (Figura 7) e sono consultabili nel percorso "*http://{host}/{port}/doc/ui*".

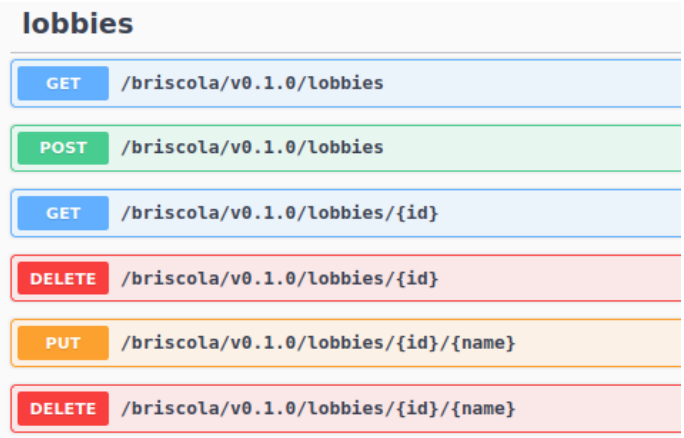

Figura 7: Swagger API docs - 1.

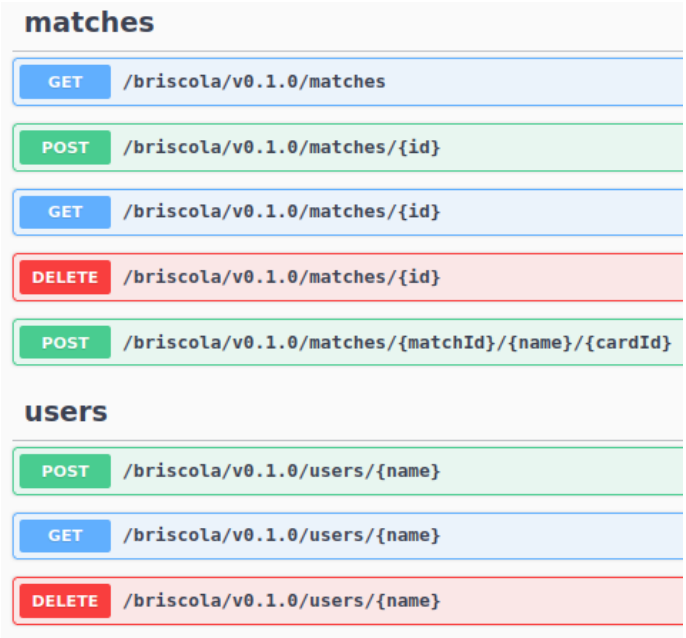

Figura 8: Swagger API docs - 2.

6 Autovalidazione

Tutta la fase di implementazione è stata accostata dal suppporto di test automatici. Tali test sono stati sviluppati con `JUnit 5`[8]. Un intero modulo del progetto (*test*) è dedicato al testing automatizzato degli altri moduli.

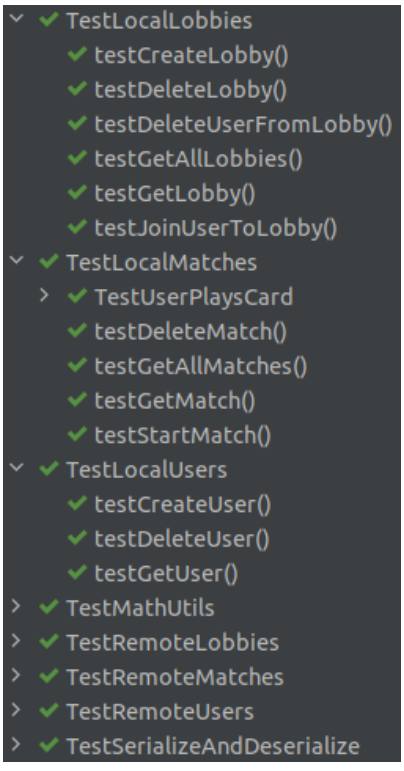

Figura 9: JUnit tests.

In particolare si è cercato di validare ogni livello di astrazione del sistema, dalla semplice comunicazione client/server alla intera logica di gioco. L'unico test effettuato manualmente è quello del corretto funzionamento della GUI. Nella Figura 9 sono mostrati i test più importanti.

Per onorare la pratica della "**Continuous Integration**", sono state sfruttate le *CI/CD Pipelines* di *GitLab*[2]. Nel dettaglio, ad ogni push di un commit su GitLab, viene creata una nuova pipeline seguendo le istruzioni nel file `.gitlab-ci.yml` (Figura 10).

In ogni pipeline vengono eseguiti due *jobs*:

- *build* → tramite il file `Dockerfile` (Figura 11), viene creato un container temporaneo dove viene lanciata un'istanza del server;
- *test* → vengono eseguiti tutti i test locali e logici, sfruttando anche il container appena creato per effettuare quelli di connettività client/server.

```yaml
image: gradle:latest

build:
  stage: build
  script:
    - cd briscola-online
    - gradle classes testClasses


test:
  stage: test
  script:
    - cd briscola-online
    - gradle test
  artifacts:
    when: always
    reports:
      junit: '**/build/test-results/test/**/TEST-*.xml'
```

Figura 10: `.gitlab-ci.yml`

```dockerfile
FROM alpine:latest
RUN apk update; apk add openjdk17
RUN apk add ffmpeg mesa-gl
COPY ./ /briscola-online/
WORKDIR /briscola-online/
RUN ./gradlew build
ENV GRD_OPTS --console=plain -q
CMD ./gradlew -p server run $GRD_OPTS
```

Figura 11: `Dockerfile`

7 Deployment

La modalità di deploy è stata oggetto di molti sunti di riflessione. L'intenzione iniziale era quella di sfruttare la containerizzazione di Docker, in quanto ciò agevolerebbe il deploy su una qualsiasi

macchina vergine con pochissimi comandi. Il problema è che l'applicazione è dotata di una GUI, perciò risulterebbe molto complesso creare un immagine docker in grado di accedere anche a mouse e monitor della macchina sul quale viene creato il container, in quanto sono necessari comandi diversi a seconda del sistema operativo ospite. Ciò comprometterebbe il principio di distribuire l'applicazione su più dispositivi possibili, a prescindere dall'OS.

La decisione finale è perciò ricaduta su **Gradle**[3]. Per poter giocare a *Briscola Online* è sufficiente che la macchina sia dotata di una JVM (versione $\geq$17). Soddisfatto questo semplice requisito, si può lanciare il server tramite il seguente comando (dopo essersi posizionati da terminale all'interno della cartella principale del repository):

```
./briscola-online/gradlew server:run
```

Questo comando lancerà un'istanza del Web Service per *Briscola Online* di default su *http://localhost:7777*. Si può specificare una porta differente come argomento al comando precedente.

Per lanciare il client di gioco, utilizzare il seguente comando:

```
./briscola-online/gradlew client:run
```

8 Esempi d'uso

Innanzitutto avviamo il server. Come mostrato in Figura 12, il server è attivo e in ascolto. Per terminarlo è sufficente chiudere la finestra.

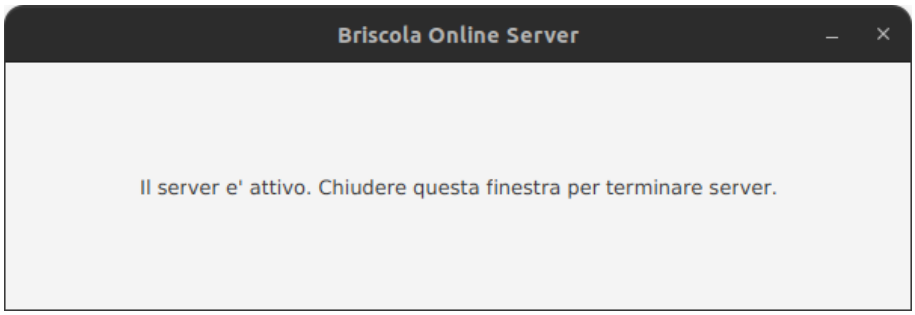

Figura 12: Server View.

Una volta avviata un'istanza del client di gioco (con il server pronto in ascolto), comparirà un piccola finestra (Figura 13) dove poter inserire il proprio nickname. Se il formato è corretto e non è già in uso da un altro utente online, si può procedere allo step successivo.

Figura 13: Login View.

Nella Figura 14 viene mostrata la *Home View* dove è possibile scegliere dall'elenco una lobby esistente alla quale unirsi oppure crearne una nuova.

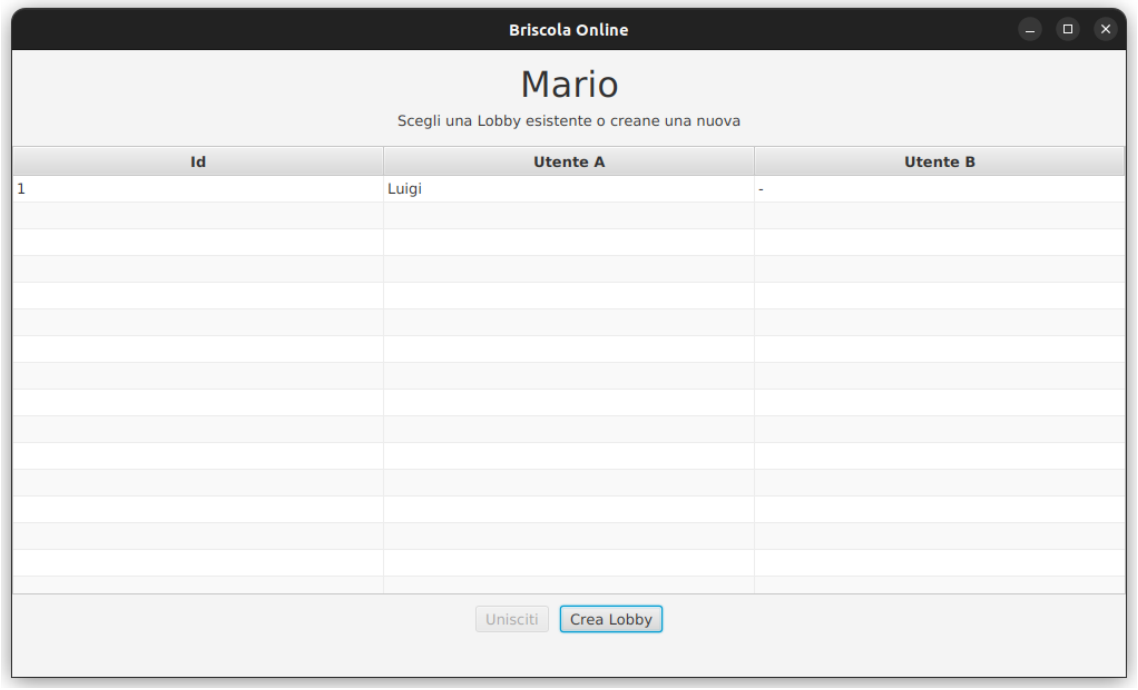

Figura 14: Home View.

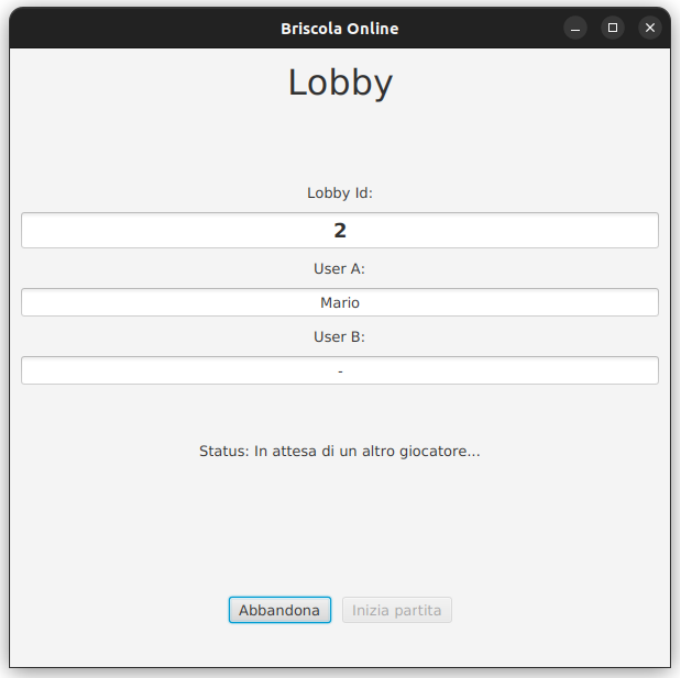

Figura 15: Lobby View - in attesa di un secondo giocatore.

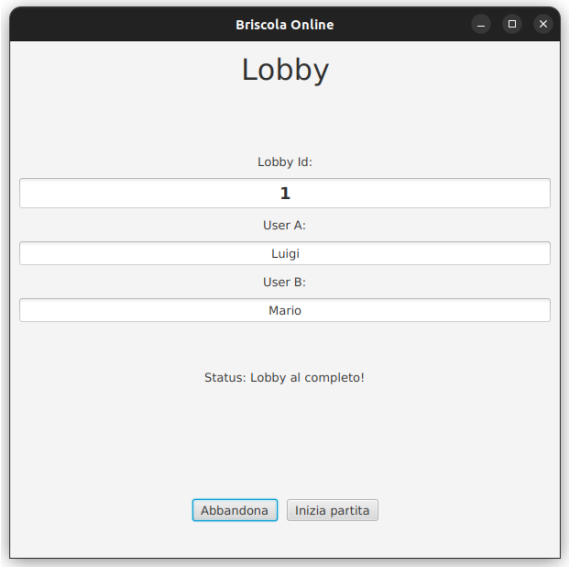

Figura 16: Lobby View - lobby al completo.

Una volta che la lobby è al completo è possibile iniziare il match.

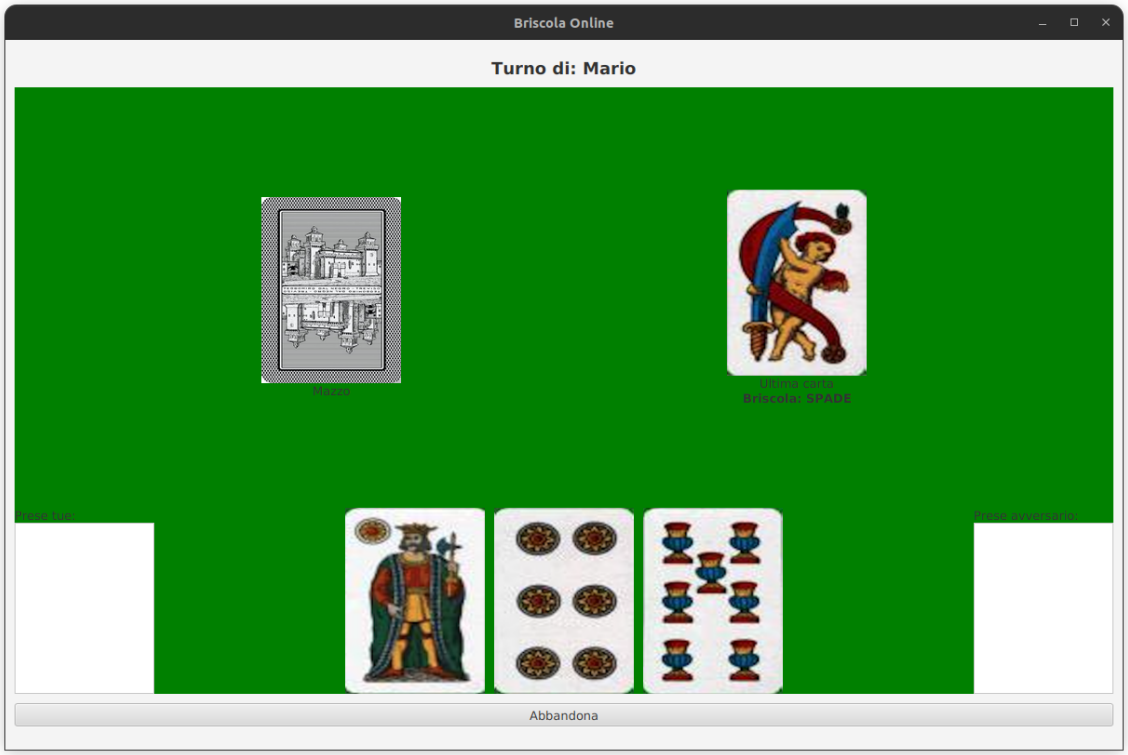

Figura 17: Match View - inizio partita.

Se è il proprio turno, si può cliccare sulla carta desiderata per giocarla, dopodichè si resta in attesa della giocata dell'avversario.

Figura 18: Match View - prima carta giocata.

A fine partità verrà mostrato il vincitore e si verrà riportati alla schermata della Lobby View (Figura 16).

9 Conclusioni

Posso ritenermi molto soddisfatto del risultato ottenuto. Sono riuscito a centrare tutti gli obiettivi fissatomi a inizio progetto. Il risultato è una versione online, scalabile, replicabile e consistente del gioco di carte di Briscola.

9.1 Sviluppi futuri

Sicuramente, i primi lavori futuri da integrare nell'aplicazione potrebbero essere:

- una modalità di autenticazione avanzata per gli utenti, con username e password;
- implementare un vero e proprio database (anzichè un semplice singleton, come nella versione attuale) per avere una persistenza dei dati. In questo modo gli utenti potrebbero ad esempio analizzare le loro partite recenti, stilare delle classifiche, ecc;
- Anche se complesso, containerizzare l'applicazione tramite Docker e fare in modo che tale container abbia accesso al monitor e al mouse della macchina, in modo da perfezionare il deploy;
- "abbellire" l'interfaccia grafica, in quanto non particolarmente curata per questo progetto (siccome non rientrava tra i focus principali del corso).

9.2 Che cosa ho imparato

Il lavoro svolto mi ha permesso di immergermi a pieno nel mondo dei Sistemi Distribuiti, apprendendo le logiche, problemi, vantaggi e svantaggi collegati a tale mondo.

Inoltre ho appreso quanto sia importante anche ragionare in fatto di *Fault Tolerance* fin dalla fase di design dell'applicazione, teamtica spesso trascurata e che viene spesso corretta nelle ultime fasi di implementazione, che porta però ad un debito tecnico considerabile.

Infine, questo progetto mi ha permesso di approfondire alcuni concetti legati alle tencologie web, nello specifico la realizzazione di un Web Service e delle sue ReST Api.

Riferimenti bibliografici

[1] *Briscola - Regole del gioco*. URL: `https://www.regoledelgioco.com/giochi-di-carte/briscola/`.

[2] *CI/CD Pipelines GitLab*. URL: `https://docs.gitlab.com/ee/ci/pipelines/`.

[3] *Gradle*. URL: `https://gradle.org/`.

[4] *Gson*. URL: `https://github.com/google/gson`.

[5] *JavaFX*. URL: `https://openjfx.io/`.

[6] *Javalin*. URL: `https://javalin.io/`.

[7] *Jetty*. URL: `https://www.eclipse.org/jetty/`.

[8] *JUnit 5*. URL: `https://junit.org/junit5/`.

[9] *Swagger*. URL: `https://swagger.io/`.