UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Ingegneria Elettrica e delle tecnologie dell'informazione

G31-T7

RemoteCCC

Professoressa:
Annarita Fasolino

Candidati:
Emanuele d'Ajello M63001435
Marco D'Elia M63001421

ANNO ACCADEMICO 2022/2023
Secondo Semestre

Indice

Capitolo 1

Introduzione

Si vuole presentare il lavoro svolto nell'ambito del corso "Software Architecture Design". Si è creato un sistema che implementa il task 7 assegnato e si è documentato tutto il processo, dalla specifica dei requisiti al testing, all'installazione. L'artefatto sviluppato è stato chiamato RemoteCCC, da ciò che il task 7 richiedeva, ovvero compilazione e copertura del codice da remoto - RemoteCompilationCodeCoverage.

1.1 Obiettivi

L'obiettivo del presente elabrato è presentare e chiarire il processo di sviluppo del sistema implementato. Si presentano documenti di varia natura per dettagliatare la creazione e l'evoluzione di tale sistema per presentare i risultati dello sviluppo.

Si vuole realizzare un sistema che permetta la compilazione, l'esecuzione remota di file di testo Java. Il primo testo include una classe java da testare, ed il secondo testo include una classe java che testa l'altre classe. Si vuole che il software produca i risultati di questa compilazione, dell' esecuzione e che calcoli la copertura del codice. Questi risultati devo essere disponibili per la consultazione dell'utente che ha richiesto il servizio.

1.2 Metodologia

Il presente lavoro è stato sviluppato in base giornaliera tramite un lavoro di team. Si è seguito uno sviluppo prototipale/incrementale. Infatti sin dalle prime fasi di sviluppo, per chiarire i requisiti e lo spazio della fattibilità, si è creato un prototipo che aiutasse il team nel capire quale fosse la direzione giusta da seguire.

Questo prototipo è stato man mano raffinato fino a diventare la vera e propria soluzione presentata in questo documento.

Le fasi di incremento del prototipo sono state scandite dalle iterazioni organizzate durante il corso "Software Architecture Design".

1.2.1 Calendario incontri

Tabella 1.1: Incontri per la progettazione del software

Data	Ora	Durata	Partecipanti	Argomento
16/4/2023	10:00	2 ore	Emanuele d'Ajello, Marco D'Elia	Brainstorming dei requisiti del software
20/4/2023	10:00	2 ore	Emanuele d'Ajello, Marco D'Elia	Secondo brainstorming dei requisiti del software
4/5/2023	14:00	4 ora	Emanuele d'Ajello, Marco D'Elia	Discussione dell'architettura del software
7/5/2023	11:00	3 ore	Emanuele d'Ajello, Marco D'Elia	Analisi dei requisiti utente
13/5/2023	9:00	3 ora	Emanuele d'Ajello, Marco D'Elia	Discussione dell'interfaccia utente
15/5/2023	16:00	2 ore	Emanuele d'Ajello, Marco D'Elia	Analisi dei rischi e delle problematiche tecniche
18/5/2023	10:00	3 ora	Emanuele d'Ajello, Marco D'Elia	Revisione dei requisiti utente
20/5/2023	14:00	2 ore	Emanuele d'Ajello, Marco D'Elia	Progettazione dell'architettura software
22/5/2023	11:00	4 ora	Emanuele d'Ajello, Marco D'Elia	Discussione sui criteri di test
25/5/2023	15:00	5 ore	Emanuele d'Ajello, Marco D'Elia	Implementazione delle funzionalità del software
27/5/2023	9:00	4 ore	Emanuele d'Ajello, Marco D'Elia	Test e risoluzione dei problemi
29/5/2023	14:00	3 ora	Emanuele d'Ajello, Marco D'Elia	Revisione del codice sorgente
2/6/2023	11:00	5 ore	Emanuele d'Ajello, Marco D'Elia	Stesura della documentazione
5/6/2023	9:00	4 ora	Emanuele d'Ajello, Marco D'Elia	Test di accettazione
8/6/2023	16:00	3 ore	Emanuele d'Ajello, Marco D'Elia	Revisione finale del software
11/6/2023	10:00	3 ora	Emanuele d'Ajello, Marco D'Elia	Consegna del software
15/6/2023	14:00	3 ore	Emanuele d'Ajello, Marco D'Elia	Valutazione dei risultati e dei feedback utente
16/6/2023	11:00	1 ora	Emanuele d'Ajello, Marco D'Elia	Pianificazione della fase di manutenzione

Tabella 1.2: Iterazioni ed incontri con professori

Data	Durata	Partecipanti	Tipo	Argomento
21/04/2023	4 ore	Marco D'Elia, Emanuele d'Ajello	Iterazione(1)	Presentazione prime scelte proggettuali
09/05/2023	4 ore	Marco D'Elia, Emanuele d'Ajello	Iterazione(2)	Presentazione prototipo
07/06/2023	1 ora	Marco D'Elia, Emanuele d'Ajello	Iterazione(3)	Raffinamento prototipo, presentazione di documentazione
10/06/2023	40 minuti	Marco D'Elia	Ricevimento	Raffinamento documentazione
14/06/2023	2 ore	Marco D'Elia, Emanuele d'Ajello	Ricevimento	Primo incontro per implementazione in università
15/06/2023	2 ore	Marco D'Elia, Emanuele d'Ajello	Ricevimento	Secondo incontro per implementazione in università

Oltre a questi incontri formalmente tabellati, ci sono stati ulteriori confronti con il gruppo 8 del task 6 al fine di chiarire interfaccie comuni per un interazione efficace.

Di seguito si è prima affrontata la fase di esplorazione dei requisiti e la loro formalizzazione e in seguito si è affrontata la fase di sviluppo tecnico.

Capitolo 2

Specifica dei Requisiti

2.1 Cos'è RemoteCCC

RemoteCCC è un servizio remoto di compilazione e copertura del codice contentente classi Java sottoposte a testing. Lo scopo di questo strumento è quello di fornire una misura dell'efficacia del test scritto attraverso il calcolo della copertura che il test genera sulla classe sotto test. Lo strumento/componente è adatto per l'utilizzo remoto e quindi può essere facilmente integrato in realtà che prediligono un'interazione a servizi.

2.2 Requisiti informali

- Il servizio di testing deve offrire agli utenti la possibilità di compilare ed eseguire casi di test per una determinata classe, inoltre il servizio deve calcolare anche la copertura di questi casi di test.
- In caso di errori di compilazione, il servizio deve restituire un messaggio, indicando le informazioni sull'errore. Se la compilazione avviene con successo, il servizio deve eseguire i casi di test, restituire l'esito dell'esecuzione, quindi la copertura ottenuta della classe sotto test.
- Il servizio deve essere semplice da utilizzare e fornire una documentazione utile per aiutare gli utenti a capire come scrivere e eseguire i casi di test.

2.3 Requisiti funzionali

I requisiti funzionali descrivono le attività che il componente deve svolgere. Nella breve descrizione; il client, colui che richiede il servizio RemoteCCC, è un attore intaregente con il sistema.

- R01: Il servizio deve offrire un'interfaccia REST per permettere al client di compilare ed eseguire casi di test.
- R02: Il servizio deve accettare due stringhe di testo, contenenti codice Java, come input: una per la classe da testare e una per la classe di test.
- R03: Il servizio deve accertarsi della correttezza sintattica delle due classi fornite. In caso di errori, il servizio deve restituire un messaggio appropriato, indicando le informazioni sull'errore (ad esempio, la riga e la colonna dell'errore nel codice sorgente). In assenza di errori, il servizio deve utilizzare un esecutore per lanciare i casi di test ed uno strumento di copertura del codice per misurare l'efficacia del test. Quindi il servizio deve ritornare al client chiamante questi risultati.

2.4 Requisiti non funzionali

- `Performance` Il servizio deve essere in grado di rispondere in un tempo medio minore di 5 secondi.
- `Manutenibilità` Il servizio deve essere facile da mantenere e da aggiornare, ovvero bisogna poter apportare una modifica significativa al software in una giornata lavorativa.
- `Compatibilità` Il servizio deve essere compatibile con ogni tipo di browser, in modo da garantire l'accessibilità del servizio per la maggior parte degli utenti.

Capitolo 3

Analisi dei Requisiti

3.0.1 Diagramma dei Casi d'Uso

La serie di requisiti funzionali del sistema trovano riscontro in un unico caso d'utilizzo: "compilazione, esecuzione e copertura". Infatti un utente richiederà sempre questo blocco di attività, senza mai poterne chiedere una specifica, ad esempio, non sarà possibile per esso richiedere la "compilazione" senza richiedere anche l'eventuale "esecuzione"; così come non sarà possibile per l'utente richiedere la "copertura" senza la "compilazione". Si è quindi scelto di rappresentare questo unico caso d'utilizzo, senza scorporarlo, per rendere evidente questa stretta connessione tra le varie attività del sistema.

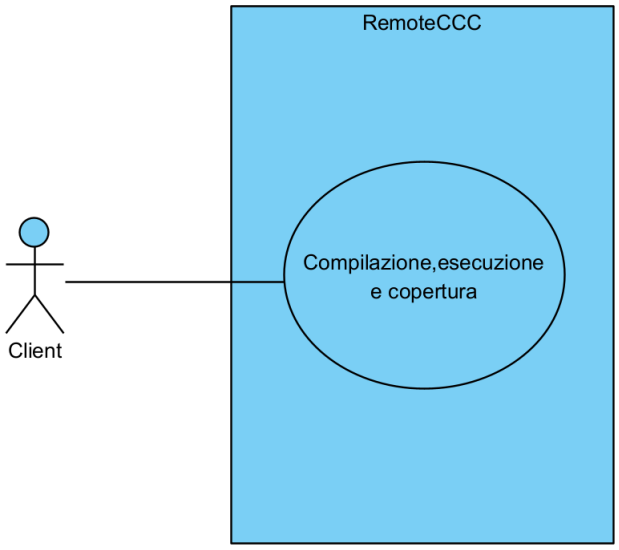

Figura 3.1: Caso d'uso

Specifica del caso d'uso "Compilazione, esecuzione e copertura"

Di seguito viene riportata la Specifica del caso d'uso:

Caso d'uso	Compilazione, esecuzione e copertura
Breve descrizione	Il sistema compila la classe da testare e quella di test, esegue il testing e ne misura la copertura.
Attori primari	Client
Attori secondari	Nessuno
Pre-condizione	il client deve poter interagire con il sistema.
Trigger	Invio di una richiesta da parte del Client verso il sistema.
Sequenza eventi principali	1) Il sistema preleva e salva le due classi fornite dal client. 2) Il sistema provvede alla compilazione delle due classi ed esegue i casi di test, contestualmente ne misura la copertura. 4) Il sistema preleva i risultati dell'esecuzione e della copertura. 5) Il sistema invia questi risultati al client.
Post-condizione	Nessuna
Sequenza eventi secondari	2b) Fallita la compilazione, il sistema non può eseguire i test e quindi misurarne la copertura. 3b) Il sistema ritorna gli errori di compilazione.

3.0.2 Diagramma di Contesto

Al fine di avere un quadro chiaro degli sviluppi implementativi che seguiranno è doveroso rappresentare sinteticamente le relazioni che ha il sistema con l'ambiente esterno, mediante l'ausilio di un diagramma di contesto.

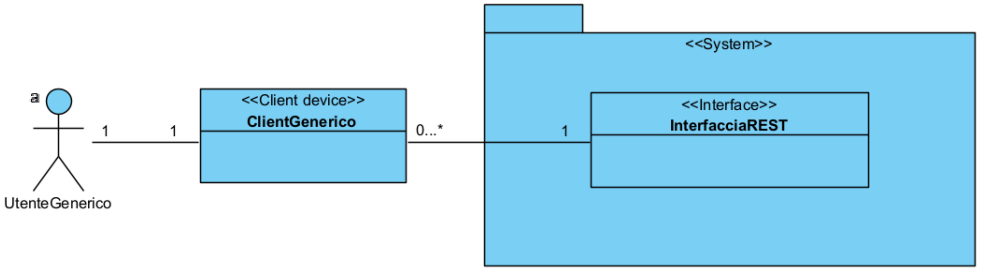

Figura 3.2: Diagramma di contesto

Il Client device è un componenete esterno al sistema RemoteCCC e che interagisce con esso per mettere in contatto il generico utente con il servizio RemoteCCC.

3.0.3 Modello di Dominio

Tramite il seguente diagramma si evidenzia la struttura ad alto livello del servizio. In una prima analisi il sistema risulta composto da una interfaccia REST, per la connessione con l'esterno, e una applicazione che si occupa di salvare, compilare, testare e eseguire la copertura delle classi passate dall'esterno.

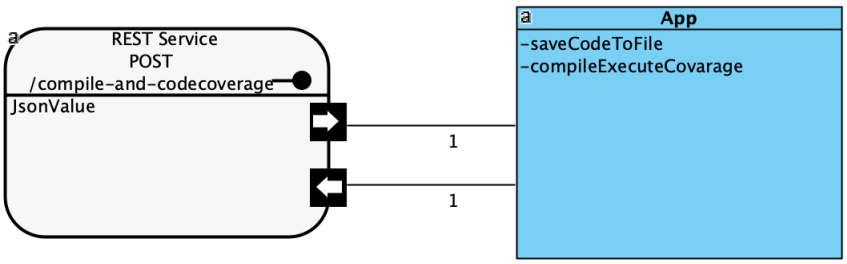

Figura 3.3: Diagramma di Dominio

3.0.4 Diagramma di Sequenza

Il seguente diagramma di sequenza mostra l'interazione tra client e sistema, quindi la temporizzazione delle varie attività. Inoltre viene fornita una chiara visione sulle alternative in esecuzione.

Si sottolinea come le attività di "compilazione, esecuzione e copertura" siano parte di un'unica attività del sistema; la ragione di questa "atomicità" verrà chiarita in seguito quando verranno introdotti i componenti del sistema.

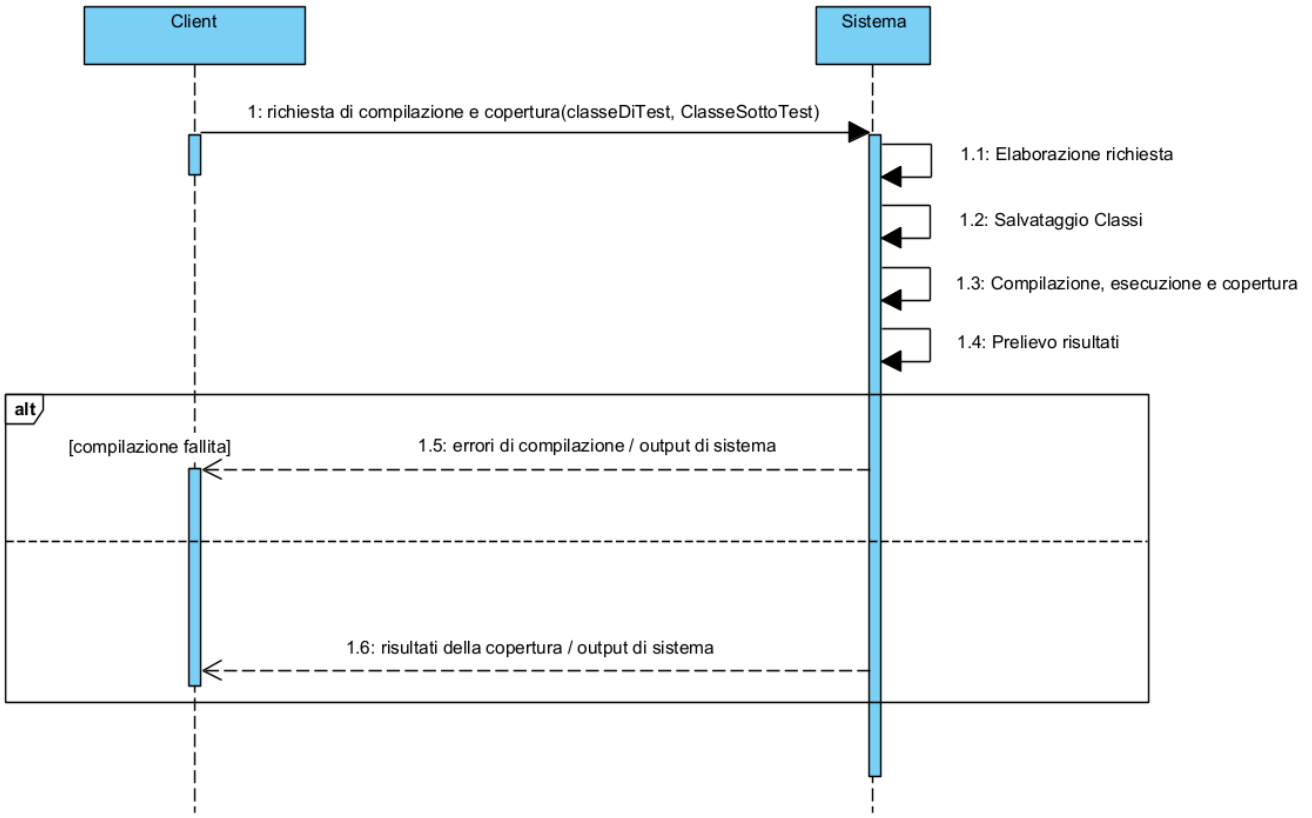

Figura 3.4: Diagramma di Sequenza

3.0.5 Architettura Preliminare

Di seguito è mostrata una prima architettura logica del sistema (che come si vedrà in seguito diventerà anche fisica, con i dovuti adattamenti) nella quale si evidenzia la struttura a pipeline, scelta data dalla natura del problema. Si evidenziano:

- Gestore delle richieste: questo componente preleva le richieste fatte dai client, interpreta la richiesta in modo che sia possibile passare al componenete successivo ciò che esso necessita.
- Compilatore/Esecutore: questo componente effettua la compilazione, l'esecuzione dei test e ne misura la copertura, da notare che queste operazioni sono interne a questo stesso componenete ed avvengono in modo atomico.
- Gestore output del sistema: questa parte del sistema (non necessariamente risultante in un componente a sè stante) effettua una operazione di prelievo ed interpretazione dei risultati forniti dalla fase precedente.

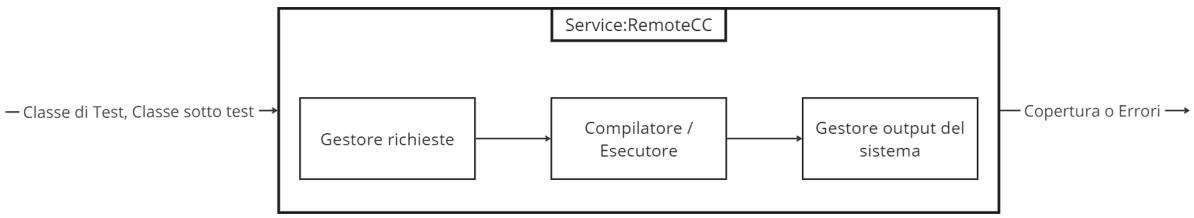

Figura 3.5: Architettura preliminare del sistema

Nel seguito del documento questa struttura concettuale si adatterà alle scelte progettuali fatte, alterandosi ma restando concettualmente valida.

Capitolo 4

Documenti di Sviluppo

In questo capitolo vengono esposte le scelte di progetto intraprese per realizzare il componenete RemoteCCC, le tecnologie utilizzate, ma anche come queste vengano tra loro integrate.

Tramite i seguenti diagrammi sarà chiaro il ruolo dei singoli componenti, cosa essi fanno, come l'applicazione li utilizza. Inoltre viene spiegato come l'applicazione interagisca con l'esteno.

4.1 Diagramma dei Componenti

Di seguito si descrivono dapprima alcuni componenenti utilizzati in RemoteCCC e poi si mostra il Diagramma dei Componenti spiegandone la ratio.

4.1.1 Descrizione di alcuni componenti

Componente SpringBoot

Java Spring è un framework open-source per lo sviluppo di applicazioni Java basate sul pattern architetturale Model-View-Controller (MVC). Il framework fornisce un'ampia gamma di funzionalità e librerie che semplificano la scrittura di applicazioni Java, riducendo la complessità del codice e aumentando la produttività degli sviluppatori. Java Spring si basa su un'architettura modulare, dove ogni modulo fornisce funzionalità specifiche. Questo approccio consente agli sviluppatori di utilizzare solo le funzionalità di cui hanno bisogno e di personalizzare il comportamento del framework in base alle esigenze del progetto. Spring può essere utilizzato per una vasta gamma di applicazioni, ed è, in questo caso, sfruttato per lo sviluppo di un'applicazione che offre un servizio RESTful.

SpringBoot é un'estensione dello Spring framework che utilizza l'autoconfigurazione per velocizzare lo sviluppo.

RESTful I servizi RESTful sono un tipo di servizio web che utilizza il protocollo HTTP per scambiare dati tra client e server. I servizi RESTful utilizzano i verbi HTTP (ad esempio GET, POST, PUT, DELETE) per definire le operazioni che possono essere eseguite sulle risorse del server.
Spring offre un supporto integrato per lo sviluppo di servizi RESTful. Gli sviluppatori possono utilizzare le annotazioni di Spring per definire i punti di ingresso del servizio, ovvero i metodi che vengono chiamati quando viene effettuata una richiesta HTTP.
Ad esempio, l'annotazione @PostMapping viene utilizzata per definire un metoodo che gestisce una richiesta HTTP POST, mentre l'annotazione @GetMapping viene utilizzata per definire un metodo che gestisce una richiesta HTTP GET.
Per implementare un servizio RESTful con Spring, gli sviluppatori definiscono un controller, che è una classe Java annotata con @RestController. Il controller gestisce le richieste HTTP, elabora i dati della richiesta e restituisce una risposta HTTP al client.

PostMapping Nella realizzazione del componente RemoteCCC è stata creata un'interfaccia POST (come si vede nel component diagram), tramite la suddetta notazione @PostMapping, all'end-point compile-and-codecoverage. Inoltre è stato specificato il tipo di body della richiesta e della risposta, ovvero Json:

```
1 @PostMapping(value = "/compile-and-codecoverage", consumes = MediaType.
      APPLICATION\_JSON\_VALUE, produces = MediaType.APPLICATION\_JSON\_VALUE)
```

Componente Maven

Apache Maven è uno strumento di gestione di progetti software che aiuta ad automatizzare il processo di compilazione, test, pacchettizzazione e distribuzione di un'applicazione. Maven è stato progettato per semplificare il processo di gestione dei progetti software, fornendo un sistema di gestione delle dipendenze, un sistema di build basato su plugin, un sistema di reportistica e un sistema di distribuzione dei pacchetti. Grazie a queste funzionalità, Maven consente di ridurre il tempo e lo sforzo necessario per configurare e gestire un progetto software complesso.

POM Maven si basa sul concetto di "Project Object Model" (POM), che definisce la struttura del progetto, le dipendenze, i plugin e le configurazioni necessarie per la sua compilazione e distribuzione. Il POM è un file XML che contiene le informazioni necessarie per il sistema di build di Maven, comprese le informazioni sul progetto, le dipendenze, i plugin, le proprietà e le configurazioni. Il POM definisce la struttura del progetto, che include la versione, il nome, la descrizione, il gruppo e l'artefatto del progetto. Inoltre, il POM definisce le dipendenze del progetto, cioè le librerie esterne necessarie per compilare l'applicazione, e le dipendenze dei plugin, cioè i plugin di Maven che devono essere eseguiti per la compilazione, il testing e la distribuzione dell'applicazione. Il POM infine può anche specificare le configurazioni dei plugin,

che definiscono il comportamento dei plugin di Maven. Ad esempio, il plugin "maven-compiler-plugin" definisce le opzioni di compilazione del progetto, come la versione di Java da utilizzare. Qui riportato il POM che permette la corretta gestione delle dipendenze del server.

Artefatti In Maven, gli artefatti sono i file binari che vengono generati durante la compilazione del progetto. Ad esempio, gli artefatti possono essere file JAR, WAR, EAR, ZIP e così via. Gli artefatti vengono creati durante il processo di build del progetto e possono essere distribuiti, archiviati o utilizzati come dipendenze di altri progetti e possono quindi essere delle dipendenze o dei plugin. Nel file POM di Maven, gli artefatti sono specificati tramite i tag "groupId", "artifactId" e "version". Questi tre elementi, insieme, identificano univocamente l'artefatto generato dal progetto.

Plugin I plugin di Maven sono una delle caratteristiche fondamentali di questo strumento di gestione dei progetti software. I plugin sono dei componenti software che vengono eseguiti durante il processo di build di Maven e che consentono di automatizzare e semplificare le diverse fasi del ciclo di vita del progetto. Un plugin di Maven può essere utilizzato per eseguire una vasta gamma di attività, come ad esempio la compilazione del codice, l'esecuzione dei test, la generazione della documentazione, la distribuzione dell'applicazione e molte altre. Grazie ai plugin, Maven consente di automatizzare queste attività e di ridurre il tempo e lo sforzo necessario per la gestione del progetto. I plugin di Maven sono implementati come componenti Java e possono essere sviluppati da terze parti o dalla comunità di sviluppatori di Maven stessa. I plugin sono distribuiti tramite un repository centrale di Maven, che consente agli sviluppatori di accedere ai plugin e di utilizzarli nei propri progetti.

Di seguito un esempio di POM:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project ...>
3
4      <modelVersion>4.0.0</modelVersion>
5      <groupId>com.example</groupId>
6      <artifactId>my-project</artifactId>
7      <version>1.0-SNAPSHOT</version>
8
9      <!-- Definizione delle dipendenze del progetto -->
10     <dependencies>
11         <!-- Dipendenza 1 -->
12         <dependency>
13             <!-- groupId -->
14             <groupId><-- groupId della dipendenza --></groupId>
15             <!-- artifactId -->
16             <artifactId><!-- ArtifactId della dipendenza --></artifactId>
```

```
17              <!-- versione -->
18              <version><!-- Versione della dipendenza --></version>
19          </dependency>
20
21          <!-- Inserire qua altre dipendenze -->
22      </dependencies>
23
24      <!-- Configurazione del plugin del compilatore Maven -->
25      <build>
26          <plugins>
27              <plugin>
28                  <!-- groupId -->
29                  <groupId><!-- GroupId del plugin --></groupId>
30                  <!-- artifactId -->
31                  <artifactId><!-- ArtifactId del plugin --></artifactId>
32                  <!-- versione -->
33                  <version><!-- Versione del plugin --></version>
34                  <!-- Configurazione del plugin -->
35                  <configuration>
36                      <!-- Configurazione del plugin -->
37                  </configuration>
38              </plugin>
39          </plugins>
40      </build>
41  </project>
```

Componente Jacoco

Jacoco è un framework di code coverage per Java. Il framework è utilizzato per analizzare il codice sorgente Java e determinare quali parti del codice sono state eseguite durante i test. Jacoco può essere utilizzato per generare rapporti di code coverage, che mostrano la percentuale di codice eseguito durante i test. Jacoco utilizza un approccio basato sulla JVM per determinare la copertura del codice. Il framework si basa sul bytecode Java generato dal compilatore Java, piuttosto che sul codice sorgente. Jacoco utilizza un agente Java che viene eseguito all'interno della JVM durante i test per determinare quali parti del codice sono state eseguite. L'agente Java registra la copertura del codice durante l'esecuzione dei test e Jacoco utilizza queste informazioni per generare i report di code coverage.

Nel capitolo API viene pure discusso l'output prodotto dall'utilizzo di Jacoco.

Di seguito è mostrato il diagramma dei componeneti:

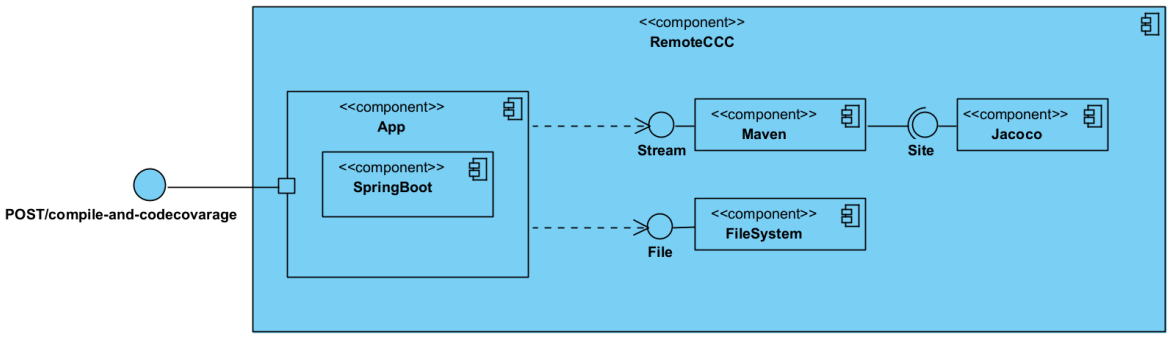

Figura 4.1: Diagramma dei Componenti

Si analizzano ora quei componeneti che ancora non sono stati trattati:

- App: il componenete App è il componenete principale dell'intero RemoteCCC, infatti esso espone l'interfaccia che poi verrà utilizzata dal client. L'App al suo interno ha il componenente, già presentato, SpringBoot; questo viene utilizzato dall'App per poter implementare tutti i servizi legati all'espositizione di una chiamata POST.
- FileSystem: questo componenete è interno al sistema operativo, ma è stato riportato nel diagramma poichè esso gioca un fuolo fondamentale della dinamica dell'App.

Inoltre si fa notare come il componente Maven venga utilizzato, come risulterà più chiaro in seguito alla spiegazione del Package Diagram, come costruttore di un progetto interno al RemoteCCC e che Maven stesso utilizzi il componenete Jacoco per misurare la copertura dei test.

4.2 Diagramma dei Package

Il seguente diagramma mostra due package principali.

- src::main::java::RemoteCCC: in questo package è mantenuta la vera e propria struttura dell'applicazione, si vedono il sottopackege App e la classe Config.
- ClientProject: questo package è stato creato per ospitare la classe sotto test e la classe da testare, creando un altro progetto interno, che possa poi essere utilizzato da Maven per eseguire i test e la misurazione della copertura. Inoltre questo package è in relazione con la classe App, infatti quest'ultima ha il compito di salvere i suddetti files, creando quindi il progetto e calcolare la copertura.
- ClientProject::src::main: qui viene salvata la classe sotto test.
- ClientProject::src::test: qui viene salvata la classe di testing.
- ClientProject::target::site: qui vengono salvati i risultati della copertura di Jacoco.

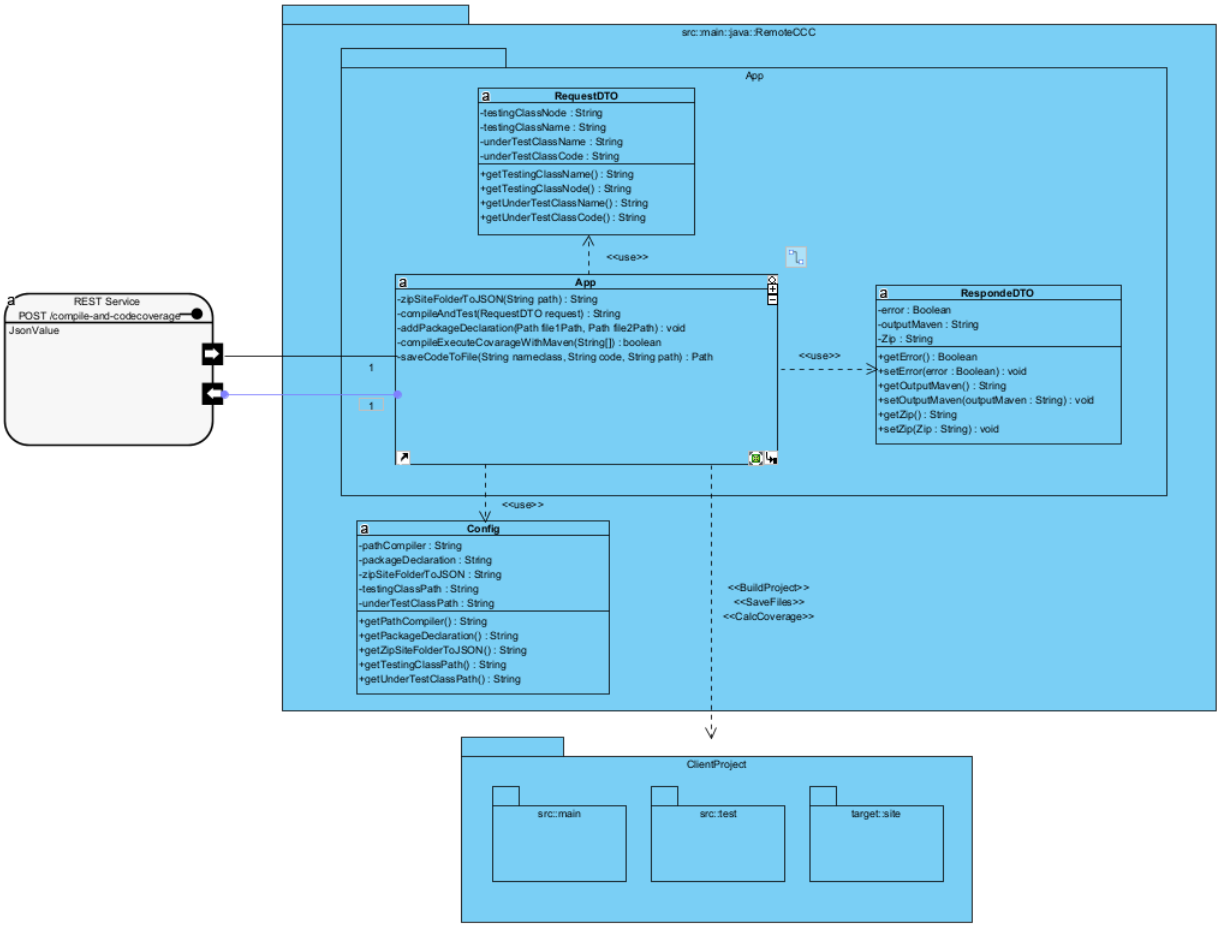

Figura 4.2: Diagramma dei Package

Di seguito vengono descritte le classi mostrate nel package:

- App: è la classe principale che coordina l'intera esecuzione, si occupa di gestire la logica del server e di tutta l'applicazione.
- Config: contiene le informazioni riguardanti i path dove salvare i file java ricevuti ed altro.
- RequestDTO: La classe "RequestDTO" è una classe statica interna che ha quattro variabili di istanza private: "testingClassName" e "testingClassCode" di tipo String, e "underTestClassName" e "underTestClassCode" di tipo String; questo oggetto è scambiato da SpringBoot all'App.
- ResponseDTO: la classe "ResponseDTO" è una classe statica interna che ha tre variabili di istanza private: "error" di tipo boolean, "outCompile" di tipo String e "coverage" di tipo String; questo oggetto è scambiato dal componente App a SpringBoot.

4.3 Diagramma delle Attività

Di seguito viene mostrato come i componenti mostrati prima interagiscano e abbiano un ruolo durante l'escuzione:

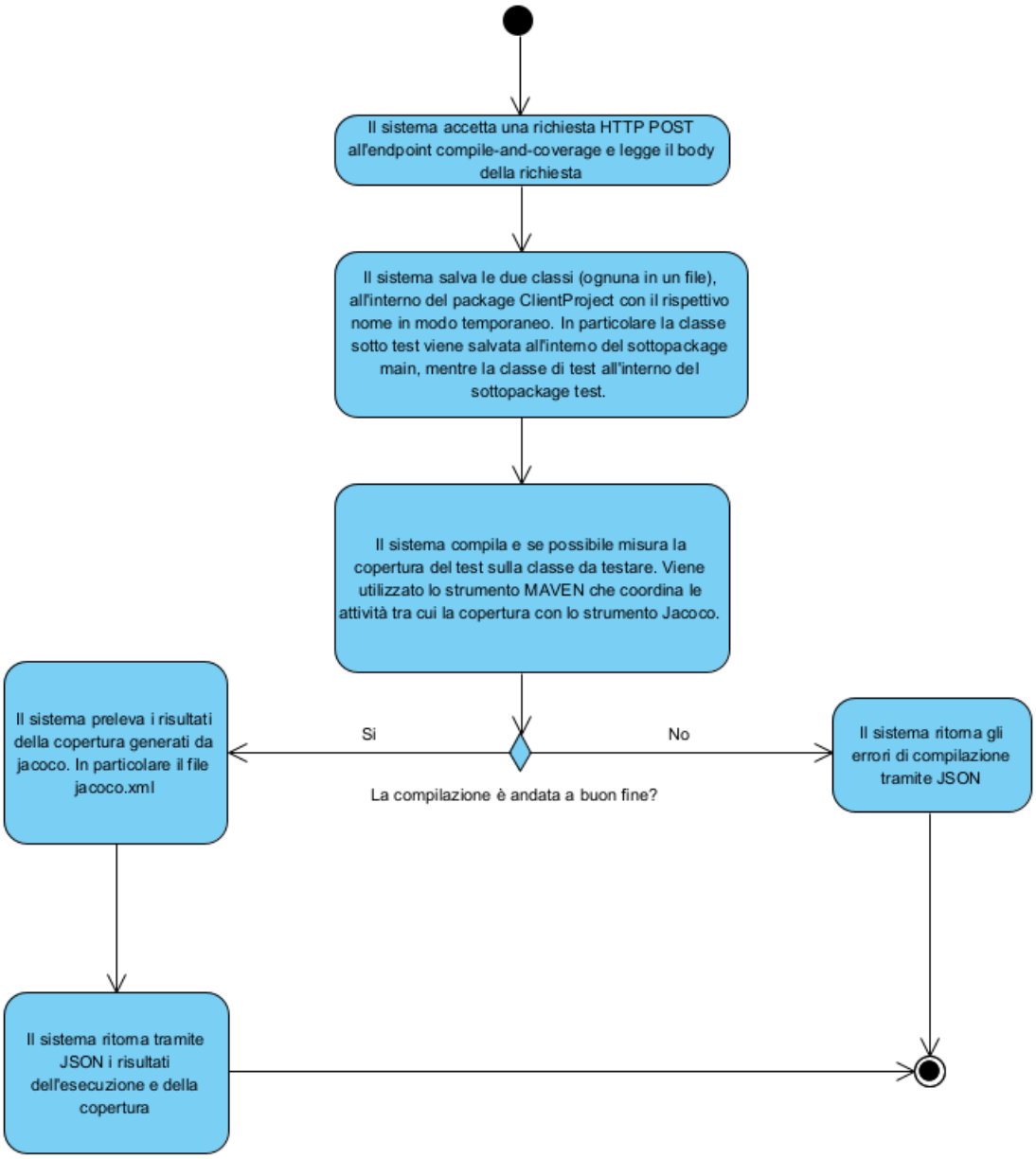

Figura 4.3: Diagramma delle Attività

4.4 Sequence Diagram di Dettaglio

A seguito è riportato il sequence diagram di dettaglio.

La prima parte mostra come il componente App si prenda carico della richiesta del Client. Una volta ricevuta quest'ultima si occupa di salvare i file sul filesystem corrente.

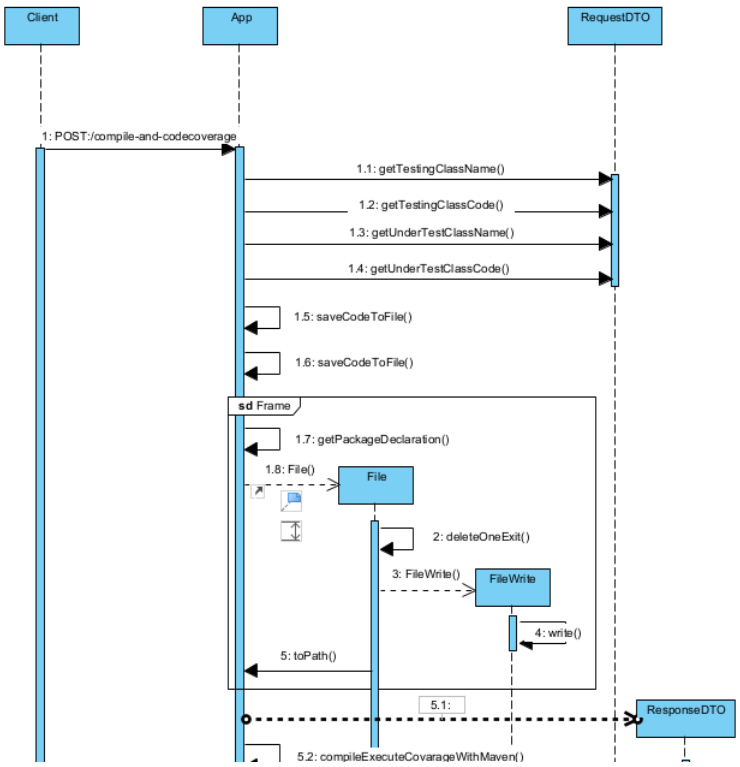

Figura 4.4: Sequence-diagram di dettaglio (1)

La seconda parte mostra come al termine della compilazione ci possono essere due risultati. Il risultato di tale operazione sarà in ogni caso riportato. A questo punto avviene la risposta al client con il data-transfer-object ResponseDTO.

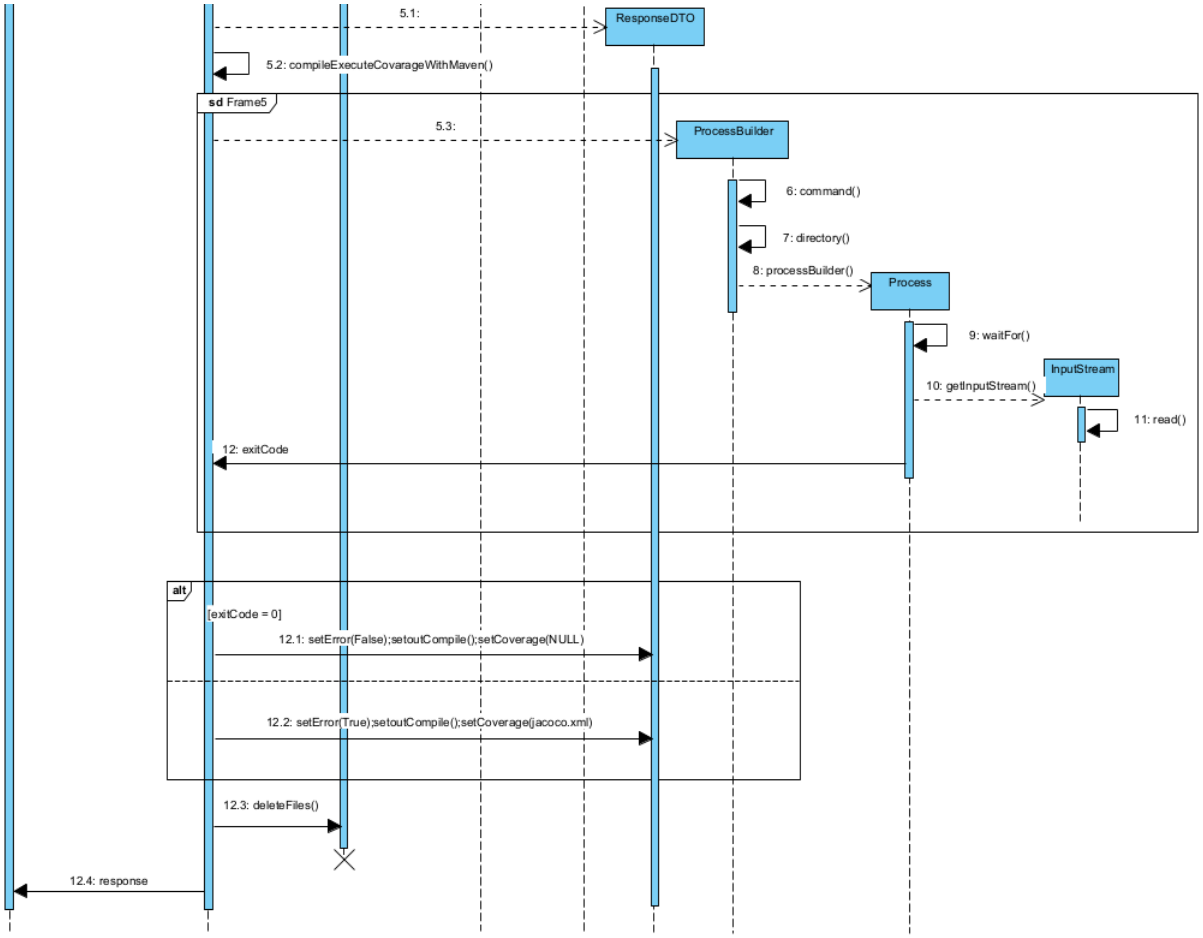

Figura 4.5: Sequence-diagram di dettaglio (2)

Capitolo 5

API

5.1 Descrizione API

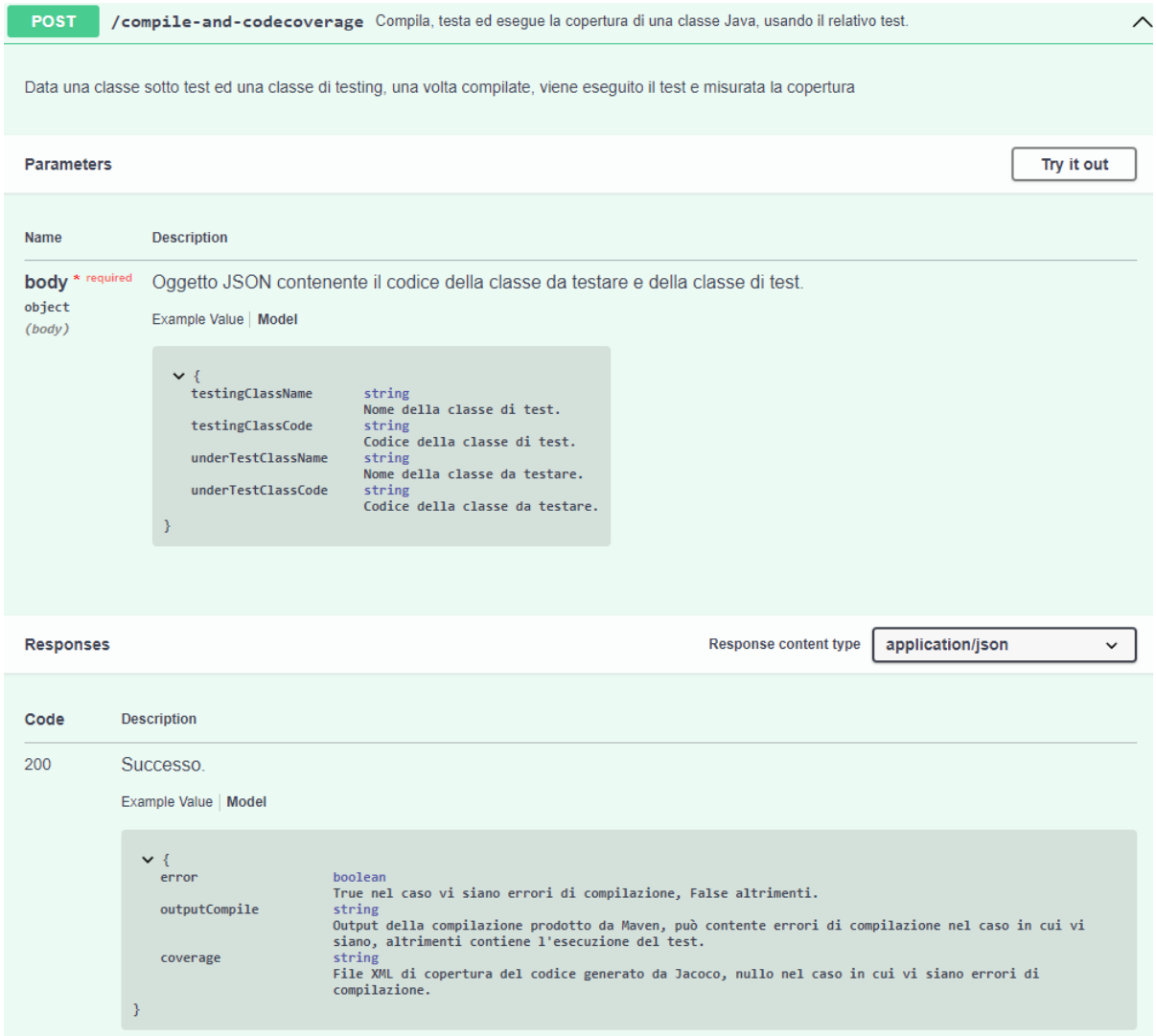

Figura 5.1: Diagramma delle Attività

Oltre al codice di risposta HTTP 200, i seguenti codici di risposta HTTP possono essere restituiti dall'API:

- **400 Bad Request** - The request was invalid or malformed.
- **500 Internal Server Error** - An unexpected error occurred on the server.

5.1.1 outCompile

Il campo outCompile è l'output su stream che viene restituito con il comando mvn clean compile test su un progetto Maven. Questo campo contiene il risultato dell'operazione di compilazione e test del progetto, fornendo informazioni sullo stato del processo e sui risultati dei test.
Il comando mvn clean compile test esegue una serie di operazioni sul codice sorgente del progetto, tra cui la rimozione dei file generati dalla compilazione precedente, la compilazione del codice sorgente e l'esecuzione dei test.
Il risultato di questo processo può essere un elenco di messaggi di errore o di successo, che indicano se la compilazione ed i test sono stati completati con successo o se ci sono stati problemi durante il processo.
Esempio:

```
1  [INFO] Scanning for projects...
2  [INFO] ------------------------------------------------------------------------
3  [INFO] Building my-project 1.0-SNAPSHOT
4  [INFO] ------------------------------------------------------------------------
5  [INFO]
6  [INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ my-project ---
7  [INFO] Deleting /path/to/my-project/target
8  [INFO]
9  [INFO] --- maven-resources-plugin:3.2.0:resources (default-resources) @ my-
       project ---
10 [INFO] Using 'UTF-8' encoding to copy filtered resources.
11 [INFO] Copying 3 resources
12 [INFO]
13 [INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ my-project
       ---
14 [INFO] Changes detected - recompiling the module!
15 [INFO] Compiling 10 source files to /path/to/my-project/target/classes
16 [INFO]
17 [INFO] --- maven-surefire-plugin:3.0.0-M5:test (default-test) @ my-project ---
18 [INFO]
19 [INFO] -------------------------------------------------------
20 [INFO]  T E S T S
21 [INFO] -------------------------------------------------------
22 [INFO] Running com.mycompany.myproject.MyClassTest
23 [INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.234 s -
       in com.mycompany.myproject.MyClassTest
```

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  10.001 s
[INFO] Finished at: 2023-06-12T15:30:00Z
[INFO] ------------------------------------------------------------------------
```

In questo esempio, il comando mvn clean compile test è stato eseguito su un progetto chiamato my-project. L'output inizia con alcune informazioni su Maven ed il progetto, tra cui la versione di Maven in uso, il nome e la versione del progetto.

Successivamente, il plugin maven-clean-plugin viene eseguito per rimuovere i file generati dall'ultima compilazione dalla directory di destinazione (/path/to/my-project/target in questo caso).

Il plugin maven-resources-plugin viene quindi eseguito per copiare le risorse del progetto nella directory di destinazione. Infine, il plugin maven compiler-plugin compila il codice sorgente del progetto e genera i file .class nella directory di destinazione.

Dopo la compilazione, il plugin maven-surefire-plugin viene eseguito per eseguire i test del progetto. Nel nostro esempio, viene eseguito un test sulla classe MyClass del progetto. Il report dei risultati dei test indica che sono stati eseguiti 5 test, tutti superati con correttamente.

In caso di errori di compilazione, le linee di output che li segnalano sono etichettate con il tag [ERROR], come mostrato nel seguente esempio:

```
[ERROR] /path/to/myproject/src/main/java/com/example/MyClass.java:[7,19] cannot
    find symbol
  symbol: class NonExistentClass
[ERROR] /path/to/myproject/src/main/java/com/example/MyClass.java:[12,9] cannot
    find symbol
  symbol: class NonExistentClass
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3
    .8.1:compile (default-compile) on project myproject: Compilation failure
```

Infine l'output mostra anche eventuali stampe eseguite nel codice.

5.1.2 coverage

Il report di copertura della code coverage di Jacoco in formato XML descrive la copertura del codice per un progetto Java. Il report contiene informazioni sul codice eseguito durante i test

per ogni classe, metodo e linea di codice.

Il report inizia con una dichiarazione XML che specifica la versione e l'encoding:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

Listing 5.1: "Dichiarazione versione ed encoding"

Inoltre il file jacoco.xml contiene una serie di elementi per rappresentare i dati di copertura. Ogni "report" JaCoCo XML è costituito da un elemento radice `<report>` che contiene uno o più elementi `<package>`. Ogni `<package>` rappresenta un pacchetto del progetto e contiene uno o più elementi `<class>` che rappresentano le classi del pacchetto.

Ogni `<class>` contiene i seguenti elementi:

- `<sourcefile>`: il nome del file sorgente della classe.
- `<name>` : il nome completo della classe.
- `<methods>`: un elenco di tutti i metodi della classe.
- `<lines>`: un elenco di tutte le linee di codice della classe.
- `<counter>`: un contatore di copertura per ogni linea di codice, che indica se la linea è stata eseguita o meno durante l'esecuzione del codice. Ogni `<counter>`: ha i seguenti attributi:
  - type: il tipo di contatore, che può essere "LINE", "BRANCH", "INSTRUCTION", "COMPLEXITY" o "METHOD".
  - missed: il numero di volte in cui il contatore non è stato soddisfatto.
  - covered: il numero di volte in cui il contatore è stato soddisfatto.

```
    <?xml version="1.0" encoding="UTF-8"?>
<report xmlns="http://www.jacoco.org/jacoco/report/v1.1.0">
  <package name="com.example.project">
    <class name="com.example.project.ExampleClass">
      <sourcefile>ExampleClass.java</sourcefile>
      <methods>
        <method name="doSomething" desc="()V">
          <counter type="METHOD" missed="0" covered="1"/>
        </method>
      </methods>
      <lines>
        <line nr="10">
          <counter type="INSTRUCTION" missed="0" covered="1"/>
          <counter type="BRANCH" missed="0" covered="1"/>
        </line>
```

```
16          <line nr="11">
17            <counter type="INSTRUCTION" missed="1" covered="0"/>
18            <counter type="BRANCH" missed="0" covered="1"/>
19          </line>
20        </lines>
21      </class>
22    </package>
23  </report>
```

Listing 5.2: Esempio di report restituito da Jacoco

5.2 Esempi di utilizzo dell'API

Introduzione Si vuole qui mostrare un esempio di utilizzo del servizio compile-and-codecoverage, in questo caso si vuole testare la classe Divide.

Classe UnderTest Per Classe UnderTest si intende il codice che si vuole testare. In questo caso si tratta di una classe che si occupa di dividere due interi, lanciando un'eccezione nel caso in cui il divisore è 0.

```
1     public class Divide {
2         public int divide(int a, int b) {
3             if (b == 0) {
4                 throw new ArithmeticException(\"Cannot divide byzero\");
5             }
6             return a / b;
7         }
```

Classe Testing La classe di testing è un codice che implementa un'instanza della classe UnderTest e ne prova le funzionalità.
Un test efficiente è un test che prova possibili valori limite per ottenere una copertura massima del codice.

```
1     import org.junit.jupiter.api.Test;
2     import static org.junit.jupiter.api.Assertions.assertEquals;
3     import static org.junit.jupiter.api.Assertions.assertThrows;
4
5     public class TestDivide {
6     @Test
7     public void testDivide() {
8         Lola cut = new Lola();
9         // Test division by non-zero number
10        int result = cut.divide(10, 2);
11        assertEquals(5, result);        // Test division by zero\n
              assertThrows(ArithmeticException.class, () -> {
```

```
12              cut.divide(10, 0);
13          });
14      }
```

Richiesta HTTP Per simulare una richiesta http/POST è possibile utilizzare il tool postman, costruendo una request che ha come url :

```
1       http://127.0.0.1:1234/compile-and-codecoverage
```

Sarà quindi lanciato sul indirizzo ip dell'host locale, utilizzando il porto 1234 ed infine l'endpoint sarà /compile-and-codecoverage.

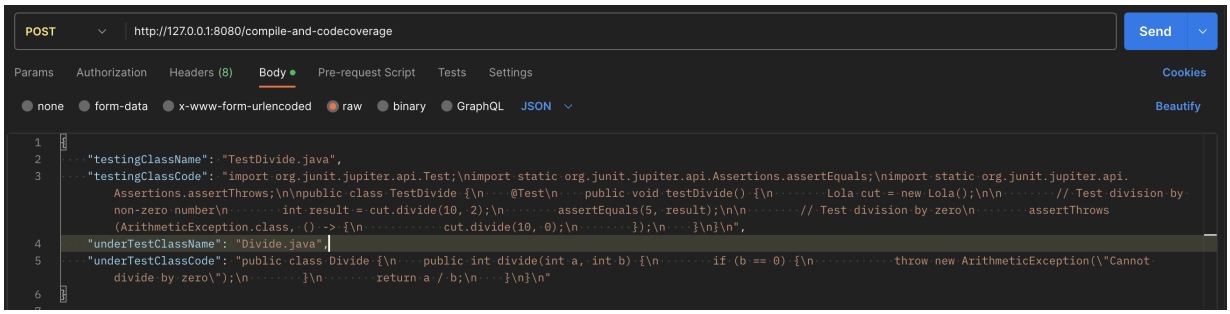

Figura 5.2: http/POST eseguita con Postman

Risposta La prima risposta è il caso in cui non ci sono errori, in questo caso: Il Json ci risponde con i tre campi error, outCompile e coverage. Ognuno di questi informa il client di quanto è avvenuto in fase di testing: Il campo error informa al client che non ci sono stati errori, il campo outCompile risponde con l'output che il processo maven ha creato, quest'ultimo è documentato e spiegato nel capitolo dedicatogli ed infine il campo coverage contiene il report di copertura generato dal framework jacoco, anche questo documentato nel capitolo dedicatogli.

Figura 5.3: Risposta priva di errori

Figura 5.4: Campo coverage

Classe di testing con errore Una risposta differente si ottiene nel momento in cui si commette un errore sintattico o logico in uno dei due codici. Infatti si è scelto di commettere volutamente un errore nella classe di test.

In questo caso la request lascia inalterata la classe sotto test, ma si è commesso un errore sintattico nella classe di test.

```
1     import org.junit.jupiter.api.Test;
2     import static org.junit.jupiter.api.Assertions.assertEquals;
3     import static org.junit.jupiter.api.Assertions.assertThrows;
4
5     public class TestDivide {
6     @Test
7     public voisd testDivide() { //ERRORE
8         Lola cut = new Lola();
9         // Test division by non-zero number
10        int result = cut.divide(10, 2);
```

```
11          assertEquals(5, result);        // Test division by zero\n
                assertThrows(ArithmeticException.class, () -> {
12              cut.divide(10, 0);
13          });
14      }
```

Risposta in caso di errori La risposta contiene nel campo error questa volta il valore true, a questo punto il client può interrogarsi sugli errori che sono stati commessi; quest'ultimi sono scritti nel campo outCompile come mostrato:

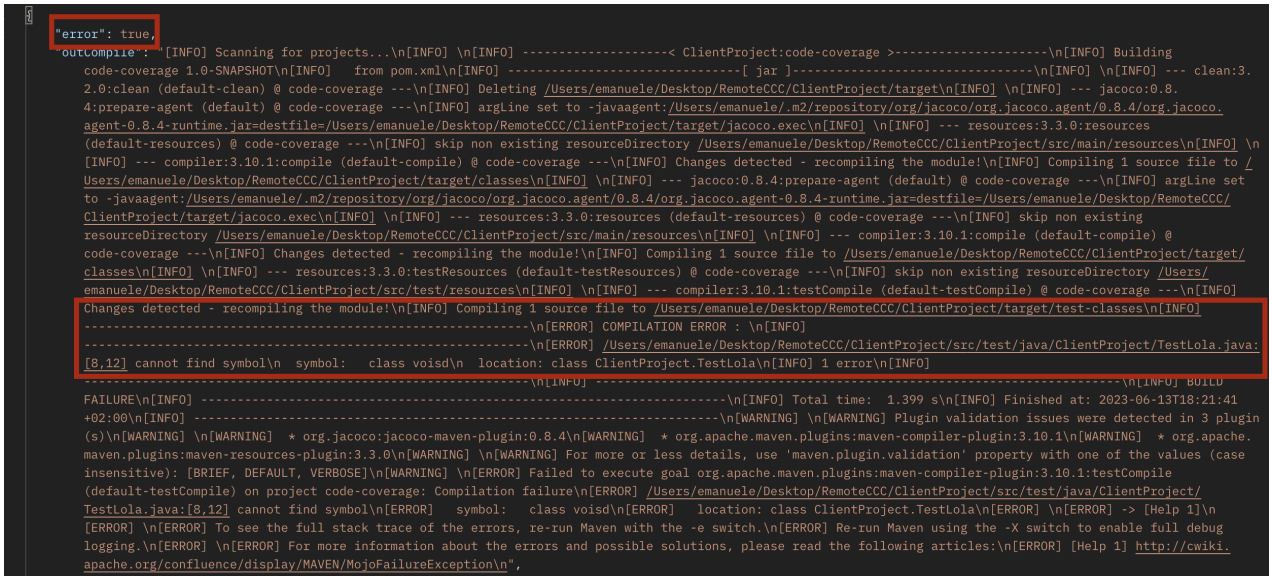

Figura 5.5: Risposta con errori

Infine il campo coverage conterrà come previsto il valore null.

Capitolo 6

Installazione ed utilizzo

6.1 Requisiti

Per installare il software è necessario possedere i seguenti requisiti:

- Apache Maven version: 3.9.2
- Java version: 17 o superiore
- Unix / MacOS ventura

6.2 Installazione ed esecuzione

Per eseguire il software basta eseguire i seguenti step:

- eseguire il git clone dall'url https://github.com/Testing-Game-SAD-2023/T7-G31.git
- spostarsi nella cartella RemoteCCC
- eseguire il comando bash: sh run.sh

L'output sarà questo:

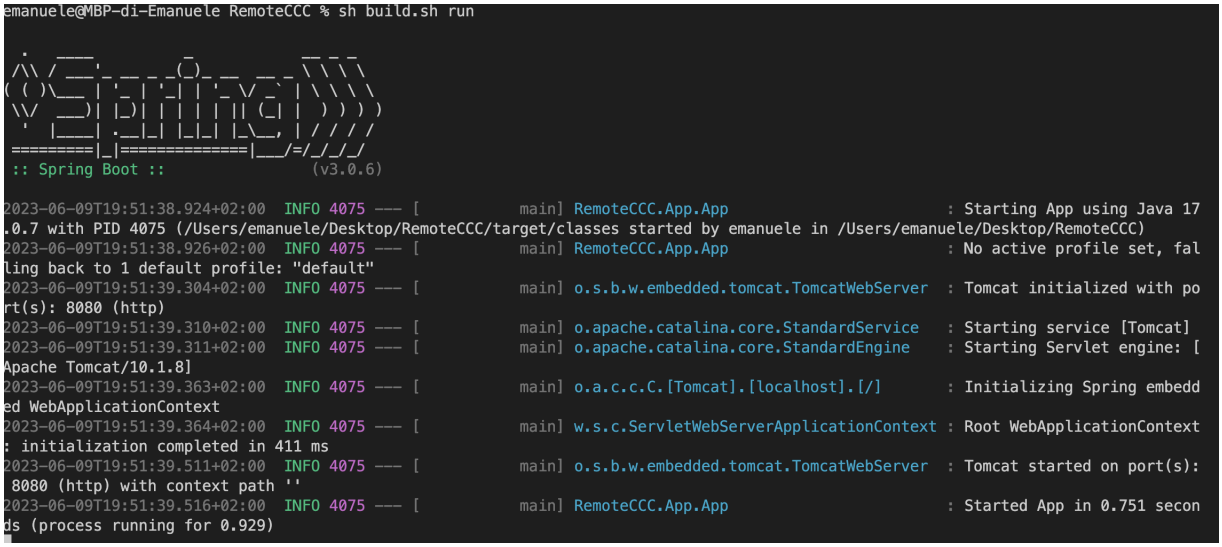

Figura 6.1: Lancio di sh build.sh run

6.3 Utilizzo

Il server è in ascolto sulla porta 1234 (impostazione configurabile come si vedrà in seguito) quindi per utilizzarlo bisogna lanciare una POST sulla porta 1234 all'end-point compile-and-codecoverage.

Di seguito si mostra un esempio di utilizzo tramite postman:

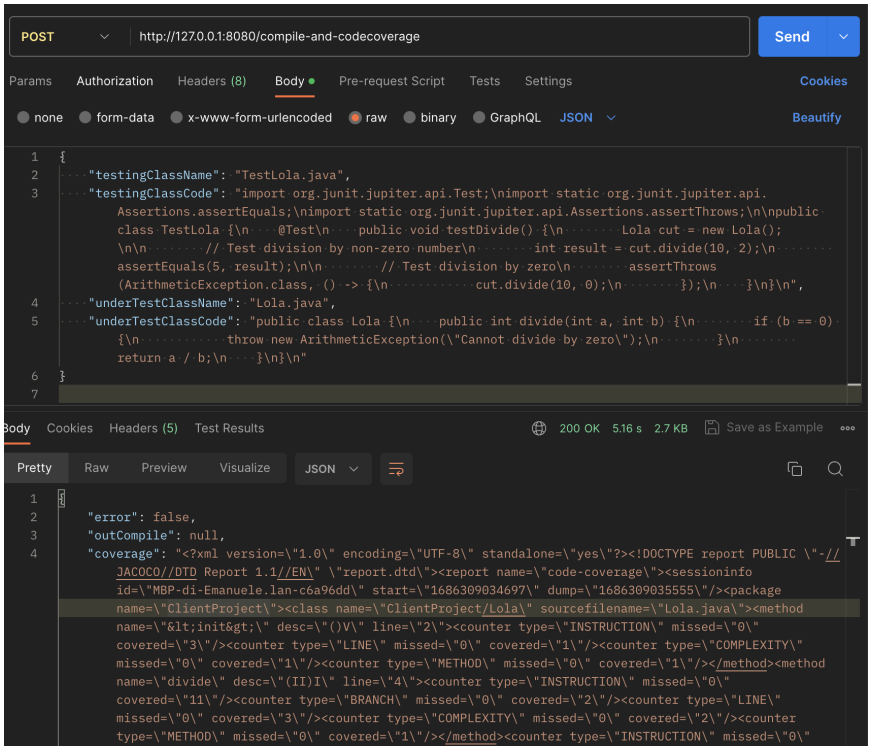

Figura 6.2: Esempio di utilizzo di postman

6.4 Configurazione

L'unica configurazione possibile è inerente alla porta sulla quale mettere in ascolto il server. Di default la porta impostata è la porta 1234.

Se si volesse cambiare la porta ed impostala alla porta "n" bisognerebbe fare:

- dalla cartella RemoteCCC, spostarsi nella cartella src/main/resources
- aprire il file application.yaml
- scrivere "n" al posto di 1234

6.5 Dipendenze

Per dipendenze si intendono qui i componenti software che all'atto dell'esecuzione di RemoteCCC, per la prima volta su di una specifica macchina, vengono scaricati. Le dipendenze si trovano nel POM del progetto e, non elencando quelli già citati, sono:

- SpringBoot
- Json versione 2023 02 27
- JaCoCo versione 0.8.2

6.6 Installation View

La vista di installazione (installation view) è una vista architetturale che descrive il processo di installazione del software su un sistema target. Questa vista è importante perché fornisce informazioni sulle risorse necessarie per l'installazione del software, software di sistema, librerie, configurazioni e dipendenze.

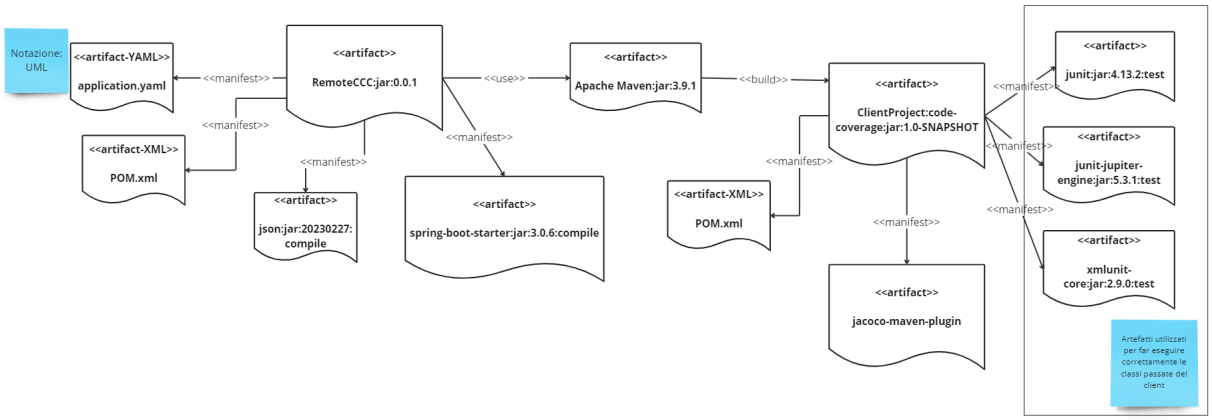

Figura 6.3: Installation view

Di seguito si spiegano i file di configurazione presenti nella installation view:

- POM.xml: questi file sono responsabili della specifica della composizione di un progetto maven, inoltre contengono tutte le dipendenze e plug-in utilizzati da questo progetto, sia in fase di compilazione che si esecuzione e test.
- application.xml: questo file contiene la specifica della porta di ascolto del server Spring Boot.

6.7 Tramite Docker

Si è scelto di spiegare l'utilizzo di docker, per l'utilizzo del componenete sviluppato, per rendere il deploy comodo e soprattutto facilmente reversibile.

6.7.1 Creazione dell'immagine g31-t7

Per creare l'immagine docker chiamata g31-t7 bisogna seguire i seguenti steps:

- scaricare tramite docker l'immagine ubuntu:latest
- scaricare git, java (17 in poi) e maven
- tramite git scaricare il progetto all'url https://github.com/Testing-Game-SAD-2023/T7-G31.git
- fare docker commit e dare il nome g31-t7

6.7.2 Utilizzo di g31-t7-container

Per utilizzare l'immagine docker g31-t7 per creare un container chiamato g31-t7-container, seguire i seguenti steps:

- avviare un docker chiamato g31-t7-container effettuando il port mapping, importante che la porta interna al containter sia quella selezionata in fase di configuarazione, di default 1234
- spostarsi nella cartella del progetto e poi nella cartella RemoteCCC
- lanciare il comando sh run.sh
- a questo punto è possibile uscire dal container e mandavi richieste

Capitolo 7

Test

I test effettuati sono stati implementati con l'ausilio di Postmann.
Si è voluto tabellare i test eseguiti esponendo quattro campi: Nome della classe di test, nome della classe sotto test, numero di righe della classe sotto test, esito della compilazione di entrambe le classi ed infine il tempo di risposta. Alcune delle classi testate sono le suguenti:

Classe test	Classe sotto test	Numero righe di test	Esito compilazione	Tempo
TestByteArrayHashMap	ByetArrayHashMap	393	Successo	2,8 secondi
TestChunkedLongArray	ChunkedLongArray	310	Successo	2.9 secondi
TestFontInfo	FontInfo	380	Successo	3.1 secondi
TestXMLParser	XMLParser	393	Insuccesso	2.0 secondi
TestRange	Range	440	Successo	4.38 secondi
TestHSLColor	HSLColor	432	Successo	2.28 secondi
TestRationalNumber	RationalNumber	400	Insuccesso	1.7 secondi
TestTimeStamp	TimeStamp	410	Successo	2.8 secondi

Risultati medi In media il tempo di risposta per singola richiesta è di 2.5 secondi per le risposte con compilazione corretta, viceversa è di 1.9 secondi. Infine si sottolinea che il tempo di risposta in caso di corretta compilazione è proporzionale al numero di righe di codice da testare.