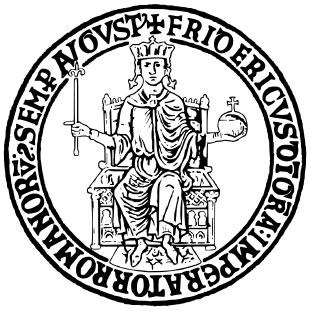
Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione



Strategie di Continuous Integration & Continuous Delivery per progetti web

Giuliano Aiello Vincenzo Tramo

2024

Indice

1 Introduzione **1**
- 1.1 Obiettivi . . . 1
- 1.2 Restful Booker Platform . . . 1
  - 1.2.1 Use Case Diagram . . . 2
  - 1.2.2 I microservizi della piattaforma . . . 3
  - 1.2.3 Il modulo `test-pilot` . . . 5

2 Automation Server: Jenkins **6**
- 2.1 Jenkins . . . 6
  - 2.1.1 Pipeline as Code e Estensibilità . . . 6
  - 2.1.2 Declarative & Scripted Pipeline Syntax . . . 7
- 2.2 Jenkins come Docker container . . . 7
  - 2.2.1 Jenkins Configuration as Code . . . 7
  - 2.2.2 Jenkins Master Worker Architecture . . . 8
  - 2.2.3 Docker in Docker (DinD) . . . 9
- 2.3 Collegamento tra Jenkins e Source Control Server . . . 10

3 Progettazione & Implementazione CI/CD **12**
- 3.1 Progettazione CI/CD . . . 12
  - 3.1.1 Build . . . 14
  - 3.1.2 Unit Tests . . . 14
  - 3.1.3 Integration Tests . . . 14
  - 3.1.4 Static Analysis . . . 15
  - 3.1.5 Build Container Image . . . 15
  - 3.1.6 Performance Tests . . . 15
  - 3.1.7 Push Container Image . . . 15
  - 3.1.8 End-to-end Tests . . . 16
  - 3.1.9 Delivery . . . 16
- 3.2 Implementazione CI/CD con Jenkins . . . 16
  - 3.2.1 Monorepo vs Multirepo . . . 17
  - 3.2.2 Struttura della repository . . . 18

3.2.3 Struttura del `Jenkinsfile` 18
3.2.4 La direttiva `agent` 19
3.2.5 La direttiva `environment` 19
3.2.6 La direttiva `options` 20
3.2.7 Il blocco `stages` 21
3.2.8 Lo `stage` 'Get Git Commit Info' 22
3.2.8.1 Dettagli sull'organization folder e Project Recognizer 23
3.2.8.2 I plugin `Git` e `Git Forensics` 24
3.2.9 Lo `stage` 'Stash Git repository' 25
3.2.10 Eseguire le singole pipeline in parallelo con `parallel` 26
3.2.11 Evitare di eseguire pipeline non necessarie 27
3.2.12 Jenkins Shared Library 28
3.2.12.1 Reference dal Jenkinsfile 28
3.2.12.2 Lo `step` custom `rbpServicePipeline` 29
3.2.12.3 Lo `step` custom `rbpSendSlackNotification` . . . 29
3.2.12.4 Lo `stage` 'Build' 30
3.2.12.5 Gli `stage` 'Unit Tests' e 'Integration Tests' 31
3.2.12.6 Gli `stage` di static analysis 31
3.2.12.7 Lo `stage` 'Build Image' 32
3.2.12.8 Lo `stage` 'Performance Tests' 34
3.2.12.9 Lo `stage` 'Push Image' 38
3.2.12.10 Il blocco `finally` - raccolta dei report 39
3.2.13 Lo `stage` 'E2E Tests' 40
3.2.14 Lo `stage` 'Delivery' 45

Conclusioni **47**

Capitolo 1

Introduzione

1.1 Obiettivi

L'obiettivo fondamentale di questa relazione è evidenziare come gli strumenti di Continuous Integration (CI) e Continuous Delivery (CD) come Jenkins possano contribuire significativamente al valore aziendale per le organizzazioni che si avventurano nella pratica dell'integrazione continua. L'essenza di questa relazione consiste nell'analizzare il processo adottato per progettare e implementare un sistema di CI/CD per un'applicazione web con un'architettura basata su microservizi.

Il progetto su cui sono state applicate le tecniche di CI/CD è una piattaforma di servizi web, di nome Restful Booker Platform (RBP), che costituisce un sistema di prenotazione per Bed & Breakfast (presentata brevemente in Sezione 1.2). È importante precisare che il progetto non è stato sviluppato da noi studenti. Tuttavia, la piattaforma rappresenta un caso di studio significativo siccome offre un terreno fertile per lo studio, la progettazione e l'applicazione di pratiche avanzate di CI/CD.

Il progetto è disponibile al seguente link GitHub https://github.com/vtramo/restful-booker-platform.

1.2 Restful Booker Platform

La piattaforma Restful Booker Platform è stata progettata per i proprietari di Bed and Breakfast (B&B) al fine di gestire i loro siti web e le prenotazioni. Il sistema offre le seguenti funzionalità:

- Creazione di un brand per commercializzare il B&B;

- Aggiungere stanze con dettagli per consentire agli ospiti di effettuare prenotazioni;
- Consentire agli ospiti di creare prenotazioni;
- Consentire agli ospiti di visualizzare dei report sulle prenotazioni per valutare la disponibilità;
- Consentire agli ospiti di inviare messaggi per contattare l'host del B&B.

1.2.1 Use Case Diagram

Lo Use Case Diagram in Figura 1.1 descrive in maniera più chiara le funzionalità del sistema:

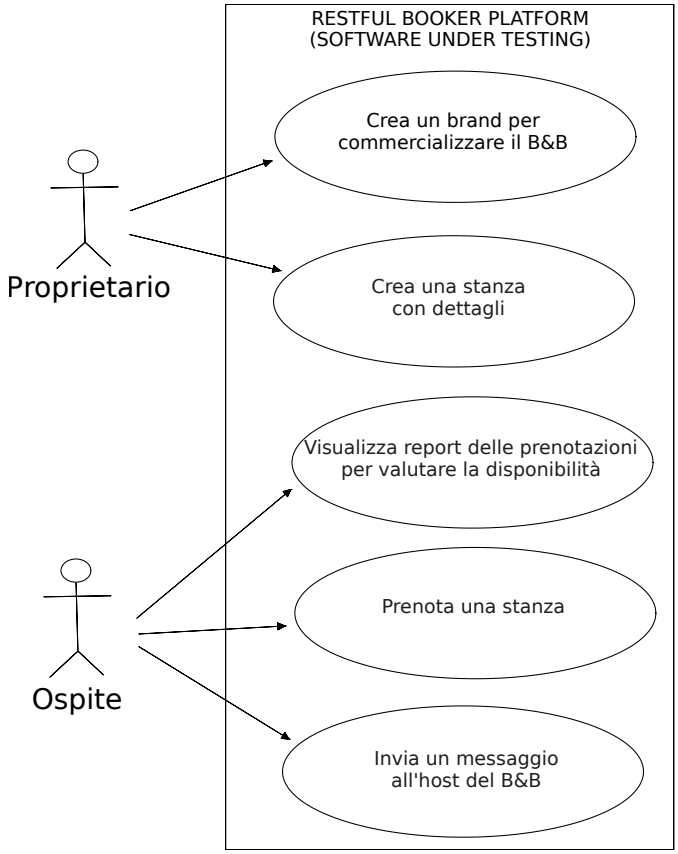


Figura 1.1: Restful Booker Platform - Use Case Diagram

L'applicazione è costruita per i B&B, il che implica la necessità di tener conto di due diverse categorie di utenti: gli ospiti e i proprietari (o hosts) dei B&B. Il

sistema comprende una varietà di funzionalità, suggerendo che più servizi sono gestiti attraverso diverse Web API.

1.2.2 I microservizi della piattaforma

La mappatura dei servizi web della piattaforma e le loro dipendenze è mostrata in Figura 1.2.

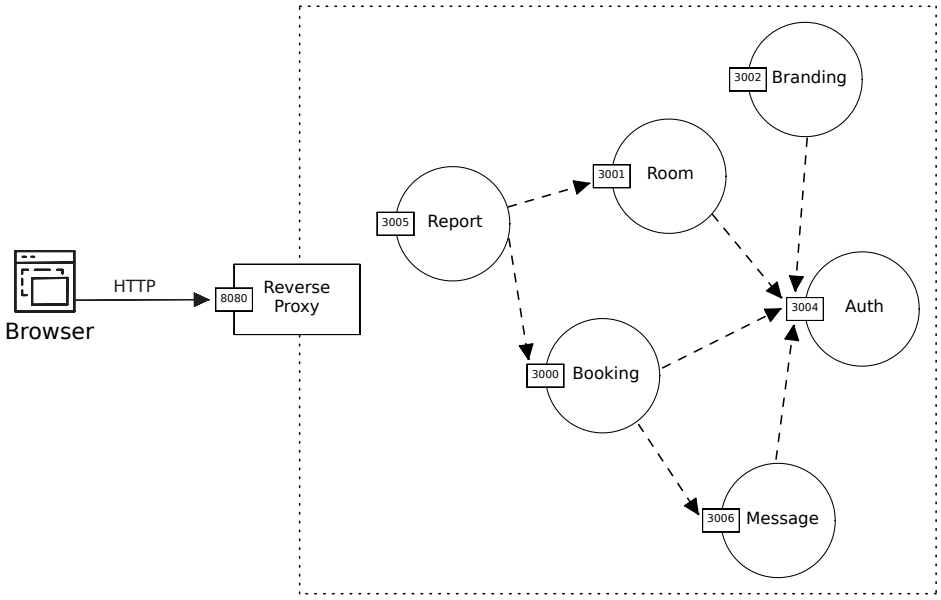


Figura 1.2: Mappatura dei servizi web di Restful Booker Platform e le loro interazioni

Ogni servizio mostrato in Figura 1.2 è una istanza **Spring Boot 2.7.5** (un popolare framework per realizzare servizi web) implementata con **Java 15**. Queste informazioni, così come i nomi dei vari servizi, sono facilmente reperibili analizzando la struttura e il codice sorgente del progetto. Di seguito vengono evidenziate a grandi linee le responsabilità di ogni servizio web:

- **Report**: questo servizio ha la responsabilità di realizzare report sulla disponibilità delle stanze. Espone due endpoint:
  1. `GET /report`, per ottenere un report di ogni stanza.
  2. `GET /report/room/id`, per ottenere un report per una specifica stanza.

- **Room**: questo servizio gestisce le stanze. Espone cinque endpoint:
  1. `GET /room`, per ottenere tutte le stanze.
  2. `GET /room/id`, per ottenere una specifica stanza.
  3. `POST /room`, per creare una stanza.
  4. `DELETE /room/id`, per eliminare una specifica stanza.
  5. `PUT /room/id`, per aggiornare una stanza già esistente.

- **Booking**: questo servizio gestisce le prenotazioni. Espone sei endpoint:
  1. `GET /booking` per ottenere tutte le prenotazioni.
  2. `GET /booking/id`, per ottenere una specifica prenotazione.
  3. `GET /booking/summary?roomId=id` per ottenere la storia delle prenotazioni di una specifica stanza.
  4. `POST /booking` per creare una prenotazione per una stanza .
  5. `PUT /booking/id` per aggiornare una specifica prenotazione.
  6. `DELETE /booking/id` per eliminare una specifica prenotazione.

- **Branding**: questo servizio gestisce la brandizzazione. Espone due endpoint:
  1. `GET /branding` per ottenere tutti i branding.
  2. `PUT /branding/id`, per aggiornare un branding.

- **Message**: questo servizio è responsabile di gestire i messaggi intesi come notifiche che un'amministratore di un B&B dovrebbe ricevere. Espone sei endpoint:
  1. `GET /message` per ottenere tutti i messaggi.
  2. `GET /count`, per ottenere il numero di messaggi non letti.
  3. `GET /message/id`, per ottenere uno specifico messaggio.
  4. `POST /message`, per creare un messaggio.
  5. `DELETE /message/id`, per eliminare un messaggio.
  6. `PUT /message/id/read`, per modificare e marcare uno specifico messaggio come letto.

- **Auth**: la piattaforma permette a un amministrazione di autenticarsi e gestire il proprio B&B. Il servizio Auth si occupa di questioni legate all'autenticazione/autorizzazione. Espone tre endpoint:
  1. `POST /auth/login` per autenticarsi e ottenere un token di accesso da inserire in ogni richiesta che lo richiede.
  2. `POST /auth/validate`, per verificare se un dato token è ancora valido o meno.
  3. `POST /auth/logout`, per effettuare il logout.
- **Reverse Proxy**: questo servizio è un reverse proxy e ha la responsabilità di smistare le richieste ai corretti servizi web della piattaforma. Riceve il traffico sulla porta `8080`.

1.2.3 Il modulo `test-pilot`

Il modulo `test-pilot` è concepito come un componente specializzato dedicato all'esecuzione di test avanzati che vanno oltre i semplici test unitari e di integrazione. La sua progettazione mira a garantire una valida infrastruttura per l'esecuzione di test sofisticati, compresi quelli end-to-end (E2E). All'interno di `test-pilot`, è possibile individuare la presenza di un'apposita directory denominata `docker`, che contiene un Docker Compose file dal nome `docker-compose-test.yaml`. Questo file è di particolare rilevanza in quanto permette di avviare l'intera applicazione Restful Booker Platform prima di eseguire i test.

Capitolo 2

Automation Server: Jenkins

Il seguente capitolo presenta una esplorazione di uno dei server di automazione più diffusi: Jenkins. Sarà delineato il processo di configurazione e avvio personalizzato di Jenkins ottimizzato appositamente per adattarsi alle esigenze della applicazione Restful Booker Platform.

2.1 Jenkins

Jenkins è un server di automazione autonomo scritto in Java e open source progettato per automatizzare una vasta gamma di attività legate al building, testing, delivering e deploying di software. È considerato uno degli strumenti di integrazione continua più popolari sul mercato, con oltre un milione di utenti.

2.1.1 Pipeline as Code e Estensibilità

Una caratteristica fondamentale di Jenkins è la possibilità di realizzare **Pipeline as Code**. La definizione di una pipeline di CI/CD è tipicamente scritta in un file di testo (chiamato `Jenkinsfile`) che a sua volta viene archiviato nella repository del codice sorgente del progetto. Questo approccio è stato utilizzato per creare degli *Jenkins jobs* capaci di compilare, testare e certificare ogni singolo cambiamento al sistema RBP. L'aspetto interessante di questo approccio è che l'intera configurazione degli Jenkins jobs può essere creata, aggiornata e controllata insieme al resto del codice sorgente dell'applicazione.

Un altro aspetto cruciale di Jenkins è la sua straordinaria estensibilità che consente di ampliarne le funzionalità attraverso l'installazione di plugin. Durante il corso di questa relazione, esploreremo alcuni di essi, focalizzandoci sui più rilevanti e significativi per il nostro contesto di studio.

2.1.2 Declarative & Scripted Pipeline Syntax

Il termine Pipeline as Code (PaC) nel contesto di Jenkins descrive un insieme di funzionalità che consentono agli utenti di Jenkins di definire processi di lavoro come se fosse codice memorizzato e versionato in un repository di origine. Queste funzionalità consentono a Jenkins di scoprire, gestire ed eseguire lavori per più repository e branch, eliminando la necessità di creare e gestire manualmente i lavori. Un `Jenkinsfile` può essere scritto usando un Domain Specific Language (DSL) basato sul linguaggio di programmazione Groovy per definire l'intero flusso di lavoro CI/CD. In particolare, le pipeline possono essere definite utilizzando due sintassi:

- *Scripted Pipeline*: questa sintassi è la sintassi originale per definire un Jenkinsfile ed è basata sul linguaggio di scripting Groovy. Il `Jenkinsfile` è scritto in Groovy e viene eseguito dal plugin Jenkins Pipeline. Questa tipologia di sintassi offre molta flessibilità e controllo sul flusso di lavoro, ma può essere più complessa e prolissa della Declarative Pipeline syntax.

- *Declarative Pipeline*: la pipeline dichiarativa è una caratteristica relativamente nuova che rende il codice della pipeline più facile da leggere e scrivere per i nuovi utenti di Jenkins. Essa si basa sempre sul linguaggio di programmazione Groovy, ma utilizza un DSL (Domain-Specific Language) basato su Groovy. Il vantaggio principale è la sua leggibilità e facilità d'uso.

È sempre preferibile definire le pipeline usando la declarative syntax. Si noti che la scripted syntax, così come la declarative syntax, non sono linguaggi di programmazione e vanno usate solo e soltanto per scopi di CI/CD.

2.2 Jenkins come Docker container

Jenkins richiede che venga ospitato su un server, il che implica la necessità di competenze infrastrutturali da parte di chi se ne occupa. La configurazione iniziale non è sufficiente; occorre prestare attenzione al sistema con aggiornamenti e manutenzione periodici per garantirne il corretto funzionamento nel tempo. In questo contesto, l'uso di un Docker container, ormai pratica comune, offre un approccio pratico e gestibile semplificando il processo di distribuzione e manutenzione di Jenkins.

2.2.1 Jenkins Configuration as Code

Parallelamente alla definizione di pipeline tramite i Jenkinsfile, è possibile formalizzare l'intera configurazione di Jenkins mediante un file YAML. Questo approccio,

noto come **Configuration as Code**, è supportato da un plugin di Jenkins chiamato Jenkins Configuration as Code (JCasC). Il file di configurazione JCasC può essere gestito tramite un controllo di versione, agevolando il tracciamento delle modifiche apportate e consentendo un ripristino a configurazioni precedenti quando necessario. E' stata avviata una specifica istanza di Jenkins containerizzata su un singolo nodo. In questo processo, è stata fornita la configurazione iniziale attraverso l'iniezione di un file di configurazione e di un documento denominato `plugins.txt`, contenente l'elenco completo dei plugin insieme alle rispettive versioni da installare durante il primo avvio dell'istanza. La Figura 2.1 sintetizza questo concetto.

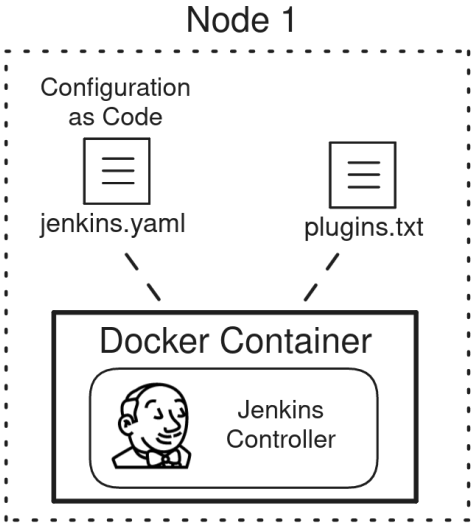


Figura 2.1: Esecuzione di un Jenkins Controller in un docker container

I file di configurazione e i Dockerfile corrispondenti possono essere trovati nella cartella `ci/jenkins` nella root del progetto.

2.2.2 Jenkins Master Worker Architecture

L'architettura di Jenkins è concepita per ambienti di build distribuiti, consentendo l'utilizzo di diversi ambienti per ciascun progetto e distribuendo il carico di lavoro tra diversi agenti che eseguono lavori in parallelo (architettura master-worker). È fondamentale distinguere tra un **Jenkins controller** e un **Jenkins agent**.

- Il controller gestisce gli agenti e coordina le loro attività, tra cui lo scheduling degli jobs sugli agenti e la supervisione delle loro prestazioni. Il ruolo del controller è gestire le connessioni e, potenzialmente, fornire gli strumenti necessari agli agenti.

- Gli agenti possono assumere forme diverse: macchine fisiche, macchine virtuali, server bare metal o Raspberry Pi. Si connettono al Jenkins controller ed eseguono attività in base alle istruzioni fornite. La tecnica che sta diffondendosi molto velocemente è, tuttavia, l'adozione di un cloud agent, un agent effimero che "nasce" on-demand su esigenza di esecuzione di un job (in altre parole, un container). Con questa tecnica, in un ambiente a grande scala, è facile immaginare i vantaggi del risparmio sulle macchine.

2.2.3 Docker in Docker (DinD)

Gli agenti Jenkins in questo progetto sono dei Docker cloud agent. Questa scelta implica la creazione di un template di un'immagine Docker che formerà, appunto, un container sul quale eseguirà un job. I job in esecuzione, come vedremo, hanno bisogno di eseguire le più comuni operazioni Docker. Dunque, su ogni Docker cloud agent viene installato un Docker client; ci si riferisce a tale procedura con il termine Docker in Docker (DinD). Al fine di consentire agli agenti Jenkins di eseguire questi comandi riferendosi a un Docker engine, e, quindi, al socket `docker.sock` della macchina host, è stato istituito un relay per trasferimenti di dati bidirezionali tra due canali di dati indipendenti: `socat`, anch'esso eseguito come container. La Figura 2.2 raffigura a grande linee questo concetto.

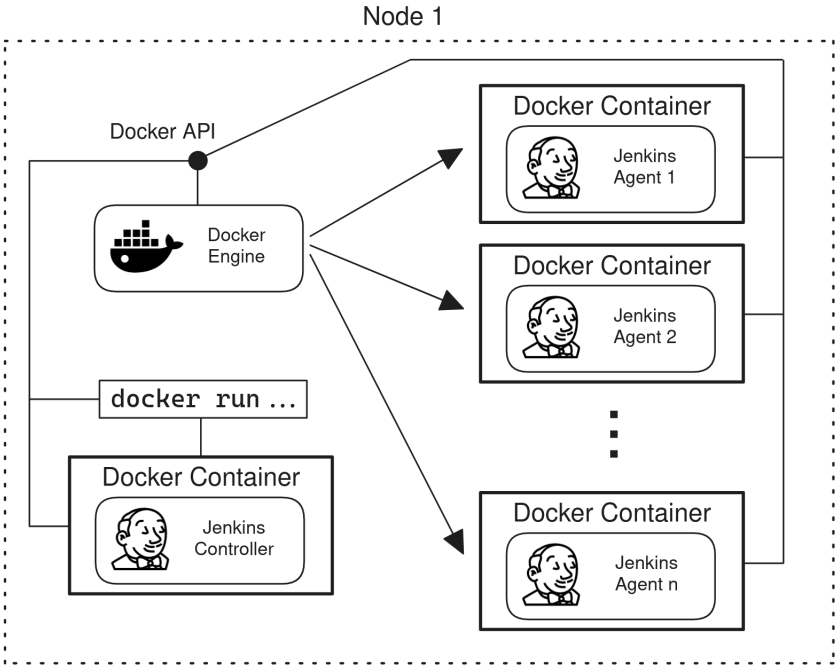


Figura 2.2: Esecuzione on-demand di agenti Jenkins

Le disparità tra gli ambienti di compilazione sono la ragione principale per cui è essenziale condurre i processi di compilazione in maniera rigorosa all'interno di container Docker. Seguendo il principio che ogni stage o azione di compilazione debba avvenire in un container Docker, si ottengono significativi vantaggi, tra cui il completo controllo dell'ambiente di compilazione, la facilità di modificare l'ambiente senza influenzare negativamente altri lavori, l'uso di versioni di strumenti esplicitamente definite e strettamente controllate, la facilità nel testare nuove versioni di strumenti, la semplificata creazione di nodi master o slave Jenkins, la capacità di eseguire qualsiasi lavoro su qualsiasi nodo e la riduzione o addirittura l'eliminazione della deriva della configurazione. Tutto ciò massimizza la flessibilità dei nostri jobs, rendendoli notevolmente più robusti.

2.3 Collegamento tra Jenkins e Source Control Server

La connessione tra Jenkins e un server di controllo del codice sorgente è un tassello inevitabile affinché possa essere progettata e implementata una (o più) pipeline di Continuous Integration e Continuous Delivery. Nel contesto di questo progetto, GitHub è stato scelto come server di controllo del codice sorgente.

Per facilitare l'interazione tra Jenkins e GitHub, è stato configurato un webhook all'interno del repository della Restful Booker Platform su GitHub. Questo notifica automaticamente il Jenkins controller ogni volta che viene effettuato un push di modifiche al repository. Il Jenkins controller, in risposta a questa notifica, attiva la pipeline predefinita, istanziando un agente Jenkins per gestire le fasi di build, test e di rilascio dell'artefatto costruito. Questa integrazione consente inoltre agli sviluppatori di effettuare modifiche al codice sorgente e di iniziare immediatamente il processo di CI/CD, ma soprattutto di ottenere immediatamente feedback sulle modifiche apportate al codice sorgente. In questo modo possono così valutare tempestivamente l'impatto delle loro modifiche migliorando l'efficienza e consentendo correzioni rapide in caso di problemi.

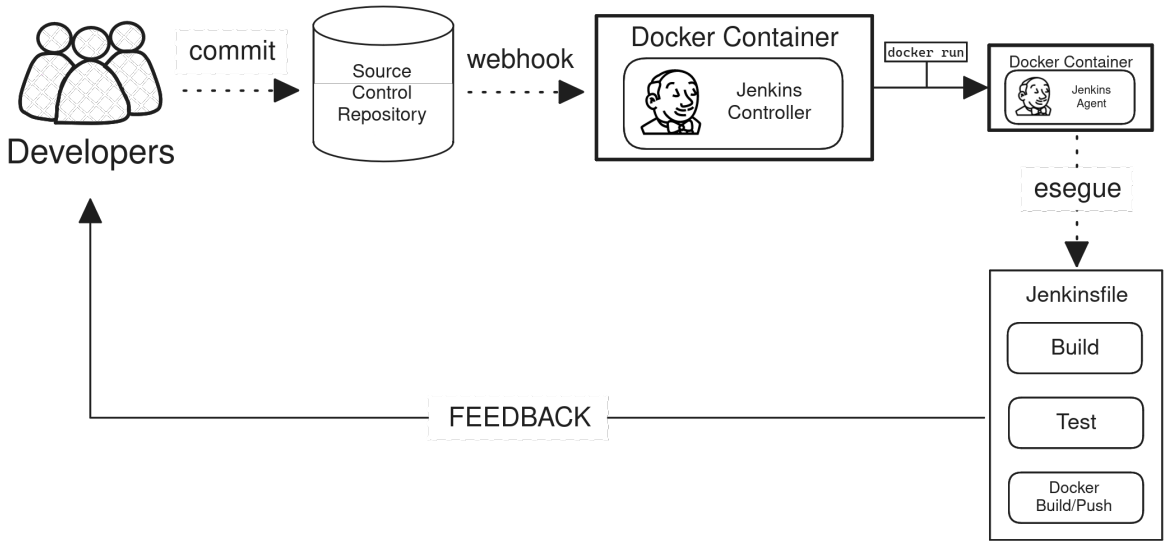


Figura 2.3: Interazione tra Source Control Server e Automation Server

Capitolo 3

Progettazione & Implementazione CI/CD

La pipeline di Continuous Integration e Continuous Delivery (CI/CD) implementata su Restful Booker Platform usando Jenkins è stata progettata per automatizzare in maniera efficiente l'intero processo di sviluppo e distribuzione del software. Questa sezione del documento approfondirà la progettazione e l'implementazione della pipeline per il progetto Restful Booker Platform (RBP).

3.1 Progettazione CI/CD

Il progetto Restful Booker Platform è principalmente una piattaforma di microservizi, ognuno dei quali svolge un ruolo specifico all'interno dell'architettura complessiva. L'essenza di questa architettura risiede nella capacità di distribuire funzionalità attraverso moduli autonomi, garantendo una maggiore flessibilità e scalabilità. Nell'ottica di garantire la coerenza e la qualità di ciascun servizio, l'obiettivo chiave è la realizzazione di una pipeline di CI/CD dedicata per ciascuno di questi microservizi. La Figura 3.1 illustra graficamente la configurazione di questa pipeline, evidenziando i flussi paralleli per ciascun microservizio. La pipeline comprende fasi distinte quali compilazione, test unitari, analisi statica, test delle performance, test di integrazione e test end-to-end (E2E). Al termine di queste fasi, viene eseguita la consegna delle immagini risultanti in un registro Docker, pronte per il rilascio in produzione. La configurazione della presente pipeline è concepita quindi con l'obiettivo di implementare la Continuous Delivery. Di seguito, esploreremo brevemente ciascuno dei singoli stage per comprendere appieno il contributo di ogni passo a questo processo di automazione e distribuzione.

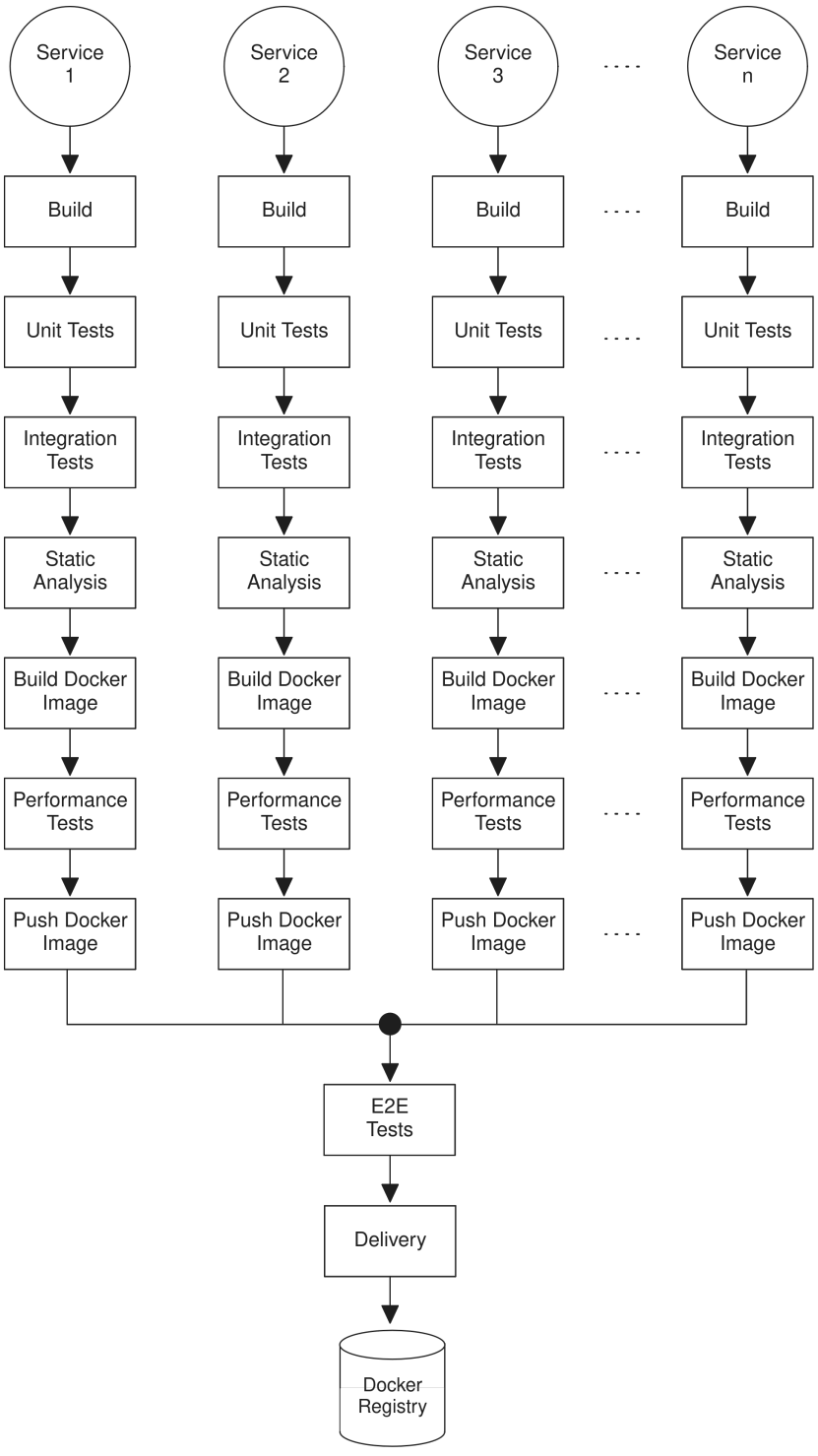


Figura 3.1: Progettazione pipeline per i microservizi di Restful Booker Platform

3.1.1 Build

Nello stage di *build*, l'attenzione è dedicata al processo di compilazione del codice sorgente del microservizio specifico, le dipendenze saranno installate e un artefatto verrà costruito. Una volta completato con successo lo stage di build, l'artefatto risultante diventa il fulcro di tutta la pipeline, fornendo la base per le fasi successive di testing e distribuzione. La decisione di posizionare lo stage di build come il primo passo della pipeline è stata guidata da un approccio "fail first" intenzionale. La priorità è garantire che qualsiasi modifica apportata al microservizio superi correttamente la fase di compilazione. Questo approccio proattivo è fondamentale per prevenire la propagazione di problemi potenzialmente costosi attraverso le fasi successive della pipeline. Questo permette di aumentare l'efficienza e l'ottimizzazione dell'uso delle risorse.

3.1.2 Unit Tests

Nella fase degli *unit tests*, la pipeline si dedica a una valutazione dettagliata della funzionalità del microservizio. Gli unit tests vengono eseguiti con un livello di granularità particolarmente fine, coinvolgendo singolarmente metodi e classi per garantire una copertura completa e accurata delle singole componenti. L'esecuzione dei test di unità è fondamentale per aumentare il grado di affidabilità, stabilità e accuratezza del microservizio. Ogni singolo test è progettato per valutare specifiche porzioni di codice cercando di coprire ogni percorso riducendo la possibilità che eventuali modifiche apportate durante lo sviluppo non abbiano avuto impatti indesiderati.

3.1.3 Integration Tests

Nella fase degli *integration tests*, la pipeline prosegue nell'analisi del microservizio in questione, spostando l'attenzione sulla sua interazione con gli altri componenti del sistema. Gli integration tests sono intesi e progettati per verificare la corretta integrazione e collaborazione tra il microservizio intero e gli altri moduli dell'applicazione. Questi test valutano la coerenza delle interfacce, la comunicazione tra i vari componenti e l'efficacia delle interazioni all'interno dell'architettura complessiva. A differenza degli Unit Tests, gli Integration Tests mirano a identificare eventuali problemi che potrebbero emergere solo quando i vari moduli interagiscono tra loro. Siccome il progetto segue una architettura a microservizi, in questa fase vengono inclusi anche i cosidetti *component tests*: test che mirano a testare solo ed esclusivamente l'intero servizio, creando, eventualmente, dei mock al posto dei microservizi su cui il servizio sotto test dipende.

3.1.4 Static Analysis

Dopo la dinamicità delle fasi precedenti, nello stage di *static analysis* si attua del testing statico: l'analisi del codice. Attraverso strumenti avanzati, è stato possibile definire delle vere e proprie regole da rispettare per gli sviluppatori, che possono portare al raggiungimento o meno di un determinato obiettivo di qualità del codice. Tutto ciò si traduce sostanzialmente nell'evitare bug, vulnerabilità, o porsi un traguardo minimo di copertura del codice testato automaticamente. Ogni obiettivo di qualità da rispettare può essere considerato come uno stage di granularità inferiore ma "pericoloso" allo stesso modo: ognuno di essi, se fallito, comporta il fallimento dell'intera pipeline.

3.1.5 Build Container Image

In questo stage viene costruita una container image utilizzando un apposito container runtime (ad esempio Docker). Tale immagine sarà usata per eseguire ulteriori test come test delle prestazioni e test end-to-end prima di essere memorizzata in un repository interno.

3.1.6 Performance Tests

Nel contesto della fase di *performance tests*, mettiamo al centro della nostra attenzione l'analisi delle prestazioni del microservizio, garantendo che possa gestire carichi di lavoro significativi senza compromettere il suo normale funzionamento. Un punto cruciale in questo stadio è l'utilizzo dell'immagine Docker precedentemente costruita e debitamente validata attraverso le fasi di build, unit tests, integration tests, e static analysis. L'adozione di un'immagine Docker semplifica notevolmente l'esecuzione dei test di performance, offrendo un ambiente controllato e coerente. In questa fase, stabiliamo obiettivi chiave di performance che il microservizio deve raggiungere, ad esempio, specifici KPI (*Key Performance Indicators*) devono rispettare determinati parametri di misurazione. Tale stage di performance tests è pensato per eseguire test di carico che simulano il quanto più possibile il comportamento dell'utente o stress tests.

3.1.7 Push Container Image

Nella fase di *Push Container Image*, il microservizio raggiunge un punto cruciale della pipeline, pronto per essere integrato e testato in scenari più complessi end-to-end insieme ad altre immagini di servizi. In questa tappa, la container image consolidata durante le fasi precedenti viene pubblicata in un repository privato,

rendendola disponibile internamente. L'immagine, frutto di una rigorosa validazione attraverso le fasi di build, test unitari, test di integrazione, static analysis e performance tests, è ora pronta per essere distribuita e integrata con gli altri componenti del sistema in un ambiente di test più sofisticato. Questo include scenari di test che coinvolgono l'interazione tra diversi microservizi, consentendo di valutare la coerenza e l'efficacia dell'intero sistema.

3.1.8 End-to-end Tests

Nella fase di *End-to-End Testing*, il focus si sposta verso una valutazione completa e integrata dell'intera piattaforma di microservizi. Questo stadio rappresenta un passo cruciale in cui tutti i microservizi vengono integrati e testati insieme per garantire il corretto funzionamento dell'architettura nel suo complesso. Durante questa fase vengono eseguiti test sofisticati che simulano scenari reali di utilizzo i quali coinvolgono l'interazione tra diversi microservizi. Gli scenari di end-to-end testing sono attentamente progettati per riflettere situazioni complesse e realistiche che la piattaforma potrebbe incontrare durante l'effettivo utilizzo. Questi test possono abbracciare flussi di lavoro completi, partendo dall'interfaccia grafica o dal browser, percorrendo l'intera catena di microservizi insieme ai suoi database individuali, fino alla risposta finale sull'interfaccia.

3.1.9 Delivery

Arrivato a questo punto, il servizio ha affrontato e superato con successo tutte le insidie messe sul percorso per rendere quanto più affidabili le funzionalità che esso espone. In questo momento, per il team, questa parte del software ha un grado di affidabilità elevato, tutti gli sforzi compiuti hanno dato esito positivo alla distribuzione del servizio. L'immagine corrente del servizio viene etichettata con una label standard che vale per ogni servizio in questo stato, la quale caratterizza l'immagine con un indicatore di sperimentazione correttamente effettuata. Essa viene in questo stesso step pubblicata sul registro privato.

3.2 Implementazione CI/CD con Jenkins

Questo capitolo esamina l'effettiva implementazione della pipeline di CI/CD utilizzando Jenkins. Dopo aver analizzato la progettazione della pipeline nei capitoli precedenti, ci concentreremo ora sull'attuazione pratica di ciascun stadio attraverso Jenkins. Verranno esplorate le configurazioni specifiche, le personalizzazioni e le soluzioni adottate per garantire un flusso di lavoro efficiente e affidabile. Nel corso

dell'implementazione, si affronteranno alcune sfide e problematiche che potrebbero emergere durante il percorso della pipeline. Saranno esaminati nel dettaglio i problemi affrontati e le soluzioni proposte per assicurare una transizione fluida attraverso ciascun stadio del processo CI/CD.

La Figura 3.2 mostra una esecuzione della pipeline in Figura 3.1 implementata ed eseguita con Jenkins.

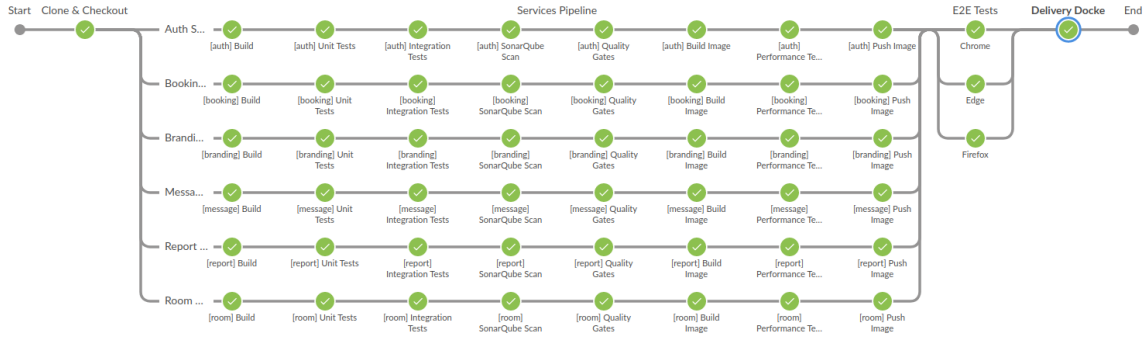


Figura 3.2: Esecuzione della pipeline in Figura 3.1 implementata ed eseguita con Jenkins

3.2.1 Monorepo vs Multirepo

Il miglior modo per integrare Jenkins con un servizio di controllo delle versioni è quello di utilizzare le cosiddette organization folders. Le organization folders consentono a Jenkins di monitorare un'intera organizzazione GitHub, un team/progetto Bitbucket, un'organizzazione GitLab o un'organizzazione Gitea. Jenkins scopre quindi automaticamente i repository Git e crea multibranch pipeline projects per ciascuno di essi, che a loro volta creano pipeline per ogni branch. Fondamentalmente, può esserci un solo `Jenkinsfile`, e quindi una pipeline, per ogni branch. Un solo file `Jenkinsfile` per branch può diventare limitante con una repository di grandi dimensioni (monorepo), dove possono esistere molteplici progetti con pipeline separate. In questo caso, il file `Jenkinsfile` può diventare molto grande e difficile da gestire.

La repository di Restful Booker Platform è una monorepo contenente differenti moduli, ognuno per ogni microservizio. Ciasciuno di questi moduli dovrebbe avere una pipeline dedicata, così come mostrata in Figura 3.1. Tuttavia, come già accennato, naturalmente viene associata a ogni repository un unico `Jenkinsfile`. Creare differenti `Jenkinsfile` in maniera tale da rispettare il flusso parallelo stile

fork/join mostrato in Figura 3.1 non è affatto banale.

Per far fronte a questo problema, è stato deciso di realizzare un unico `Jenkinsfile` posizionato alla radice del progetto.

3.2.2 Struttura della repository

Di seguito viene riportata la struttura della repository di Restful Booker Platform.

```
.
|-- assets/
|-- auth/
|-- booking/
|-- branding/
|-- message/
|-- proxy/
|-- report/
|-- room/
|-- test-pilot/
|-- ci/
|-- Jenkinsfile
|-- docs/
|-- docker-compose.yml
|-- pom.xml
|-- build_locally.cmd
|-- build_locally.sh
|-- run_locally.cmd
|-- run_locally.sh
|-- README.md
```

La repository contiene 9 moduli, tra cui 6 sono dei microservizi su cui bisogna definire una pipeline come quella in Figura 3.1. Si noti che la repository contiene soltanto un `Jenkinsfile`, precisamente nella root del progetto.

3.2.3 Struttura del `Jenkinsfile`

Così come accennato in 2.1.2, è sempre preferibile utilizzare la declarative syntax. Una declarative pipeline deve iniziare con `pipeline` come mostrato di seguito:

```
pipeline {
    // Esegui pipeline per ogni servizio coinvolto nella modifica

    // Alla fine, dopo che tutte le pipeline sono state eseguite
```

```
        // con successo, esegui i test e2e

        // Consegna le nuove immagini
    }
```

L'esempio appena esposto è esattamente ciò che si trova nel `Jenkinsfile`.

3.2.4 La direttiva `agent`

La direttiva `agent` viene utilizzata all'interno di una pipeline per specificare dove eseguire gli *step* o gli *stage* della pipeline. Il `Jenkinsfile` di RBP immediatamente specifica un `agent`:

```
    pipeline {
        agent {
            label 'build-agent'
        }

        ...
    }
```

La direttiva `agent` specifica che gli step successivi della pipeline dovrebbero essere eseguiti su un agente Jenkins contrassegnato con l'etichetta `'build-agent'`. Un'etichetta è una stringa associata a un agente Jenkins specifico. Gli agenti con l'etichetta corrispondente saranno selezionati per eseguire il lavoro definito nella pipeline. L'agente etichettato `'build-agent'` è una immagine Docker contenente tutti i tool necessari per eseguire gli step della pipeline, ciò è stato affrontato in maniera teorica nel paragrafo 2.2.3.

3.2.5 La direttiva `environment`

Proseguendo nel `Jenkinsfile` di RBP, troviamo il blocco `environment`:

```
pipeline {
    agent {
        label 'build-agent'
    }

    environment {
        DOCKER_REGISTRY_URL = credentials('docker-registry-url')
    }
```

```
    ...
}
```

La direttiva `environment` viene utilizzata all'interno di una pipeline per definire variabili d'ambiente che saranno disponibili durante l'esecuzione della pipeline. In questo caso, la direttiva è utilizzata per impostare una variabile d'ambiente chiamata `DOCKER_REGISTRY_URL` nel contesto della pipeline. La variabile `DOCKER_REGISTRY_URL` viene poi impostata utilizzando il valore restituito da una credenziale chiamata `'docker-registry-url'` opportunamente creata sul Jenkins controller. Tale variabile sarà disponibile in ogni parte della pipeline e in qualsiasi nuovo ambiente di esecuzione (anche in un nuovo docker container successivamente creato).

3.2.6 La direttiva `options`

La direttiva `options` viene utilizzata per configurare opzioni globali o comportamenti specifici della pipeline.

```
pipeline {

    ...

    options {
        disableConcurrentBuilds()
        buildDiscarder logRotator(artifactDaysToKeepStr: '',
                                  artifactNumToKeepStr: '100',
                                  daysToKeepStr: '',
                                  numToKeepStr: '100')
        timestamps()
        skipStagesAfterUnstable()
        parallelsAlwaysFailFast()
        timeout(time: 7, unit: 'MINUTES')
    }

    ...
}
```

È possibile impostare diverse opzioni che influenzano l'esecuzione complessiva della pipeline. Nel `Jenkinsfile` di RBP vengono definite globalmente le seguenti opzioni (mostrate sopra):

- `disableConcurrentBuilds()`: questa opzione assicura che solo un'istanza di questa pipeline possa essere eseguita contemporaneamente. Se una pipeline

è già in esecuzione, un'altra attesa verrà messa in coda fino a quando la precedente non è completata. Questa opzione evita che due build eseguite concorrentemente o in parallelo possano interferire tra loro. Tuttavia, in alcuni casi, la concorrenza potrebbe essere desiderabile.

- `buildDiscarder logRotator(...)`: questa opzione specifica come gestire i vecchi artefatti di build. Nel `Jenkinsfile` di RBP si utilizza un `logRotator` per conservare 100 build più recenti senza limiti di tempo. I file di log possono occupare uno spazio significativo, specialmente se la pipeline viene eseguita frequentemente o se produce un output dettagliato. Limitando il numero di build mantenute, si evita di saturare lo spazio su disco con file di log non più necessari.

- `timestamps()`: aggiunge i timestamp alle console di output della pipeline, fornendo informazioni su quando ogni passaggio è stato eseguito.

- `skipStagesAfterUnstable()`: se uno dei passaggi della pipeline non è andato a buon fine, questa opzione impedisce l'esecuzione di passaggi successivi. Questo permette di evitare di eseguire gli stage successivi.

- `parallelsAlwaysFailFast()`: la pipeline esegue in parallelo le pipeline di ogni microservizio in quanto queste sono indipendenti tra loro. Questa opzione fa sì che la pipeline interrompa immediatamente se uno qualsiasi dei rami paralleli fallisce.

- `timeout(time: 7, unit: 'MINUTES')`: imposta un timeout globale per l'intera pipeline. Se la pipeline non è completata entro 7 minuti, verrà interrotta. Impostare un timeout è essenziale in quanto uno step della pipeline potrebbe rimanere bloccato per qualsiasi ragione. In questo caso, se nessuno interrompe manualmente la pipeline, la build corrispondente rimarrà per sempre attiva consumando risorse prezione. I timeouts rilevano tempi di processamento anormali. È meglio essere notificati di una build fallita a causa di un timeout che essere notificati il giorno dopo.

3.2.7 Il blocco `stages`

Il blocco `stages` viene utilizzato per definire una serie di fasi o passaggi (`stage`) che compongono la pipeline. Ogni stage rappresenta un'unità di lavoro specifica, e il blocco `stages` consente di organizzare la pipeline in modo ordinato e chiaro.

```
...

stages {
```

```
        stage('Get git commit info') {
            ...
        }

        ...
    }

    ...
```

Il blocco `stages` esegue sequenzialmente (se non specificato esplicitamente una esecuzione parallela) gli `stage` in esso contenuto. Nel `Jenkinsfile` di RBP il primo stage è quello di ottenere informazioni sul commit responsabile dell'avvio della pipeline.

3.2.8 Lo `stage` 'Get Git Commit Info'

Ottenere informazioni sul commit Git all'interno di una pipeline rivela la sua utilità in diversi contesti. Per prima cosa, consente di tracciare chi ha eseguito il commit, quando è stato eseguito e quali modifiche sono state apportate al codice sorgente. Questo aspetto è cruciale per garantire una chiara tracciabilità del processo di sviluppo.

Nel contesto del versioning e del release management, avere un riferimento diretto al commit in cui è stata introdotta una modifica diventa fondamentale. Semplifica notevolmente il tracciamento delle versioni del software e la gestione delle release, offrendo una visione chiara dell'evoluzione del codice nel tempo.

Inoltre, l'associazione del commit alla build specifica di Jenkins che ha prodotto l'artefatto è di particolare importanza. Questa connessione permette di risalire agevolmente alla build corrispondente e di esaminare in dettaglio tutti i logs, i risultati dei test e altri output generati durante l'esecuzione della build. Tale approccio facilita il processo di troubleshooting e analisi post-build, fornendo un quadro completo delle attività svolte durante l'esecuzione della pipeline.

Ancora, come vedremo, le informazioni sul commit possono essere utilizzate per generare notifiche o report automatici. Ad esempio, è possibile inviare automaticamente un'email o un messaggio di notifica contenente i dettagli del commit a team specifici o a utenti interessati.

Lo stage 'Get git commit info' nel `Jenkinsfile` di RBP è il seguente:

```
...

stage('Get git commit info') {
    steps {
```

```
            script {
                env.GIT_COMMIT = sh (script: 'git rev-parse HEAD',
                                     returnStdout: true)
                env.GIT_SHORT_COMMIT = "${env.GIT_COMMIT[0..7]}"
                env.GIT_PREVIOUS_SUCCESSFUL_SHORT_COMMIT =
                    env.GIT_PREVIOUS_SUCCESSFUL_COMMIT[0..7]
                env.GIT_COMMITTER_NAME =
                    sh (script: "git show -s --format='%an' ${env.GIT_COMMIT}",
                        returnStdout: true)
                env.GIT_COMMITTER_EMAIL =
                    sh (script: "git show -s --format='%ae' ${env.GIT_COMMIT}",
                        returnStdout: true)
                env.GIT_COMMIT_MSG =
                    sh (script: "git log --format=%B -n 1 ${env.GIT_COMMIT}",
                        returnStdout: true)
            }
        }
    }

    ...
```

Questo blocco `stage` denominato 'Get git commit info' è progettato per ottenere informazioni cruciali sul commit Git corrente associato alla build corrente. L'uso di questi passaggi consente di catturare informazioni chiave sul commit, come l'hash completo e abbreviato, il committer, l'email e il messaggio del commit. Queste informazioni possono essere successivamente utilizzate all'interno della pipeline per scopi come la generazione di notifiche, la documentazione dei rilasci o il tracciamento delle informazioni associate a ciascun processo di build. Si noti il blocco `script`. In tale blocco abbiamo il consenso di utilizzare la scripted syntax e utilizzare il linguaggio di programmazione Groovy. La variabile `env` è un puntatore alle variabili d'ambiente correntemente definite nella build corrente che Jenkins fornisce automaticamente.

3.2.8.1 Dettagli sull'organization folder e Project Recognizer

Nel Jenkins controller è stata istituita una organization folder, la quale è stata opportunamente collegata a un'organizzazione GitHub. Attraverso questa connessione, Jenkins è in grado di individuare dinamicamente tutte le repository associate a tale organizzazione GitHub e genera automaticamente un multibranch pipeline project per ciascuna di esse. Nel contesto specifico, l'organizzazione GitHub in questione ospita un'unica repository denominata `restful-booker-platform`,

dando vita a un singolo multibranch pipeline project. L'essenza di un multibranch pipeline project è la sua capacità di rilevare automaticamente ogni branch all'interno della repository corrispondente. In questa implementazione, considerato che l'unico branch disponibile è `trunk`, viene quindi generata una pipeline unica. Affinché questa integrazione operi in maniera sinergica, è necessario configurare un webhook nell'organizzazione GitHub che punti direttamente al Jenkins controller. Ogni volta che si verifica un commit all'interno della repository `restful-booker-platform`, GitHub attiva il webhook, notificando di conseguenza il Jenkins controller. Quest'ultimo, a sua volta, avvia la build della pipeline associata. Tuttavia, affinché il Jenkins controller possa individuare correttamente la posizione del `Jenkinsfile` all'interno della repository, è fondamentale associare un project recognizer all'organization folder. Questo componente definisce in modo preciso il percorso relativo all'interno della repository in cui si trova il `Jenkinsfile`, permettendo al sistema di eseguire la pipeline.

3.2.8.2 I plugin `Git` e `Git Forensics`

Un multibranch pipeline project è già associato a una repository Git. Per questo motivo, Jenkins automaticamente esegue il clone e il checkout di tale repository, creando un *workspace* corrispondente. Affinché tutto ciò possa funzionare, il Jenkins controller deve contenere il plugin Git. Il plugin git fornisce le operazioni git fondamentali per i progetti Jenkins. Può eseguire polling, fetch, checkout, branch, list, merge, tag e push dei repository.

Oltre al plugin Git, sul Jenkins controller è stato installato il plugin Git Forensics. Il plugin Git Forensics è progettato per offrire analisi approfondite sulla storia del repository Git associato al progetto. Esso consente di esplorare e comprendere la dinamica degli sviluppi nel tempo, fornendo informazioni dettagliate su chi, quando e cosa ha contribuito al codice sorgente. Gli utenti possono visualizzare statistiche sulle attività di commit, esaminare cambiamenti specifici all'interno del repository, e ottenere una comprensione completa delle dinamiche di sviluppo. Questo strumento facilita la valutazione della produttività del team, l'identificazione di pattern di sviluppo, e l'individuazione di potenziali punti di interesse nella storia del codice. Per permettere questa analisi approfondita, bisogna "minare" la repository Git. Per fare ciò, è stato aggiunto uno stage 'Mine Repository'.

```
...

stage('Mine repository') {
    steps {
        mineRepository()
    }
```

```
    }

    ...
```

Questo `stage` usa lo step `mineRepository()` fornito dal plugin Git Forensics per eseguire una analisi approfondita della build. Alla fine, dopo tale stage, sull'interfaccia grafica del Jenkins controller troviamo dei grafici e dati da analizzare come quello in Figura 3.3.

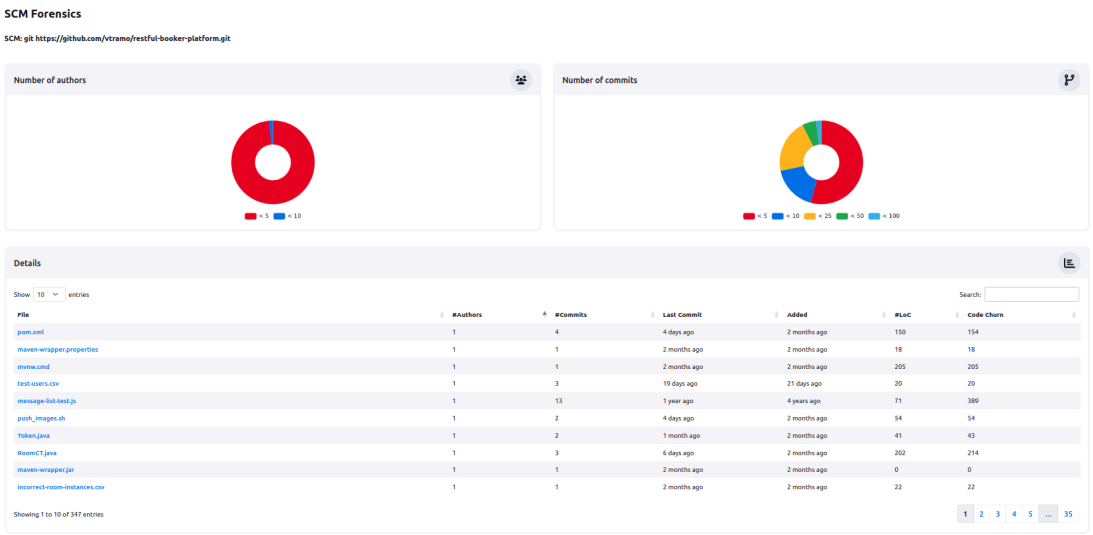


Figura 3.3: Il plugin Git Forensics

3.2.9 Lo `stage` 'Stash Git repository'

Ogni pipeline di ogni microservizio viene eseguita nel contesto di un *nuovo* docker container (un nuovo Jenkins agent). Questo permette di eseguire il lavoro associato al particolare servizio in isolamento, senza interferire con le altre pipeline che, come già accennato, eseguono in parallelo. Tuttavia, siccome i docker container sono effimeri, tutti gli artefatti fino ad adesso prodotti dagli `stage` precedenti non saranno trattenuti. In particolare, la Git repository non sarà disponibile. Per evitare di clonare nuovamente la repository `restful-booker-platform`, è possibile utilizzare gli step `stash` e `unstash`. Con questi step, Jenkins permette di salvare un insieme di file da utilizzare successivamente su qualsiasi nodo/agent/-workspace nella stessa build. I file archiviati con `stash` sono archiviati in un file TAR compresso. Il seguente `stage` nel `Jenkinsfile` esegue esattamente questa operazione:

```
...

stage('Stash git repository') {
    steps {
        stash includes: '**', name: 'rbp', useDefaultExcludes: false
    }
}

...
```

Con tale step stiamo dicendo di "mettere da parte" l'intera repository clonata. L'oggetto così archiviato ha il nome `rbp`. I nuovi ambienti di esecuzione possono quindi ottenere velocemente (rispetto al clone) la repository archiviata eseguendo `unstash`:

```
unstash 'rbp'
```

3.2.10 Eseguire le singole pipeline in parallelo con `parallel`

Come già accennato e da come è possibile dedurre dalla Figura 3.1, le singole pipeline dei singoli servizi possono essere eseguite in parallelo perché indipendenti. Per tradurre questo nella sintassi dichiarativa di Jenkins, creiamo uno stage di nome 'Services Pipeline' che esegue più `stage` in parallelo:

```
...

stage('Services Pipeline') {
    parallel {
        // esegui stage 'Auth Service'
        // esegui stage 'Room Service'
        // esegui stage 'Booking Service'
        // esegui stage 'Branding Service'
        // esegui stage 'Message Service'
        // esegui stage 'Report Service'
    }
}

...
```

Ogni `stage` all'interno del blocco `parallel` verrà eseguito in un nuovo docker container evitando così ogni tipo di interferenza tra le singole pipeline in esecuzione. L'esecuzione di ogni `stage` all'interno di un docker container non è una caratteristica della direttiva `parallel` ma bensì un dettaglio implementativo della nostra pipeline che sarà più chiaro successivamente.

3.2.11 Evitare di eseguire pipeline non necessarie

Quando una modifica ha influenzato soltanto un microservizio dovrebbe essere eseguita *solo e soltanto* la pipeline di quel microservizio. Questa pratica mirata si traduce in un'ottimizzazione delle risorse e un aumento dell'efficienza, in quanto consente di concentrare gli sforzi di compilazione, test e distribuzione solo sul microservizio coinvolto dalla modifica. In tal modo, si minimizzano i tempi di esecuzione complessivi.

Per raggiungere questo obiettivo, possiamo usare la direttiva `when`. La direttiva `when` consente alla pipeline di determinare se lo `stage` deve essere eseguito in base alle condizione date. Le nostre condizioni sono di eseguire la pipeline solo se c'è stata una modifica al codice sorgente, ai test o al `Dockerfile` di *quel* microservizio. Per ciascuno degli `stage` relativi ai vari microservizi, introduciamo un blocco `when` che consente di valutare queste condizioni specifiche:

```
...

stage('Services Pipeline') {
    parallel {
        stage('Auth Service') {
            when {
                anyOf {
                    changeset "auth/Dockerfile"
                    changeset "auth/src/main/java/**/*.java"
                    changeset "auth/src/test/java/**/*.java"
                }
            }

            // Esegui pipeline per il servizio auth
        }

        ...

    }
}

...
```

In questo caso, la pipeline di 'Auth Service' (così come le pipeline individuali di tutti gli altri servizi) verrà eseguita solo se la condizione fornita nel `when` risulta vera.

3.2.12 Jenkins Shared Library

Facendo riferimento sempre alla figura 3.1, è facile intuire che ogni pipeline di ogni singolo servizio presenta esattamente la stessa struttura. Per evitare ripetizione di codice e seguire invece il concetto DRY, anche in questo caso ci si affida a Jenkins che mette a disposizione la possibilità di creare delle *Shared Libraries*, cioè delle repository esterne (rispetto a quella esistente su cui è definita la pipeline) con la possibilità di referenziarle, e poter usufruire delle funzioni di cui esse dispongono. Seguendo le indicazioni Jenkins, la seguente è la struttura della repository creata per ospitare la Shared Library messa a disposizione per essere richiamata nel Jenkinsfile di RBP. La cartella `vars` ospita i file script che sono esposti poi come `steps` nella pipeline che utilizza la libreria, proprio attraverso il nome del file; nella cartella `src` risiedono sostanzialmente delle classi, che possono servire da utility alla libreria.

```
.
|-- src/
  |--BuildStatus.groovy
  |--BuildUtilities.groovy
  |--Service.groovy
  |--SlackUtilities.groovy
|--vars/
  |--rbpSendNotification.groovy
  |--rbpServicePipeline.groovy
|-- README.md
```

Prima di analizzare nel dettaglio alcuni dei file elencati, è bene precisare che per utilizzare questa libreria all'interno del `Jenkinsfile` di RBP è stato necessario configurarla sulla dashboard di Jenkins specificando le modalità di importazione, nonché il link alla repository[1].

3.2.12.1 Reference dal Jenkinsfile

Come appena detto, il Jenkinsfile deve avere la possibilità di usufruire della funzioni libreria, il collegamento quindi tra queste due entità lo si è ottenuto specificando all'inizio del Jenkinsfile il nome della libreria già configurata sul Jenkins controller `@Library('rbp-jenkins-shared-library') _`.

Lo sviluppo di uno `stage` appartenente ad un service come quello presentato in precedenza, segue un template adottato da ogni servizio (evidente conseguenza dell'uso giustificato delle Shared Library) che corrisponde a tutti gli effetti ad una chiamata della funzione della Shared Library. Ogni stage effettuerà la

[1] https://github.com/vtramo/rbp-jenkins-shared-library

chiamata specificando i propri parametri, ad esempio, la chiamata effettuata dal servizio `room` si presenta nel seguente modo, lo scopo delle variabili sarà chiaro successivamente:

```
...

stage('Room Service') {
    when{
        ...
    }
    steps {
        rbpServicePipeline(
            serviceName: 'room',
            nodeLabel: 'build-agent',
            rbpServiceHostname: 'rbp-room',
            rbpServicePort: '3001',
            skipPerformanceTests: true
        )
    }
}

...
```

3.2.12.2 Lo `step` custom `rbpServicePipeline`

Nella struttura della repository elencata nel paragrafo 3.2.12, tra tutti i file, andremo ad analizzare attentamente `vars/rbpServicePipeline.groovy`. All'interno di esso sono contenuti i "veri" passi della pipeline che agiscono sul service. Come accennato in 2.1.2, attraverso una `label` dev'essere indicato il nodo su cui la pipeline verrà eseguita, ci avvaliamo quindi della variabile d'ambiente mostrata in 3.2.12.1. Notiamo che pur essendo in realtà sempre lo stesso agent uguale per tutti i service, manteniamo una variabile d'ambiente per far sì che in un futuro, il cambio di agent da parte di uno dei service sia più agevolato, favorendo molto la manutenzione.

Facendo invece riferimento al paragrafo 3.2.9, la prima cosa che viene eseguita all'inizio è proprio l'istruzione volta ad ottenere in modo veloce ed efficace la repository su questo agent. Il codice continua poi con una struttura `try-catch`, ciò che segue farà parte del blocco `try` nel file citato, fino a quando diversamente specificato.

3.2.12.3 Lo `step` custom `rbpSendSlackNotification`

All'interno della Shared Library, è stato implementato un ulteriore `step` perso-

nalizzato denominato `rbpSendSlackNotification()`. Questo particolare `step` è stato progettato per inviare notifiche personalizzate a un canale Slack specifico, informando tempestivamente l'intero team del risultato della build. Grazie a questa notifica automatizzata, il team può rimanere prontamente informato sui successi e gli eventuali fallimenti delle build. La Figura **??** mostra una notifica di terminazione positiva di una build.

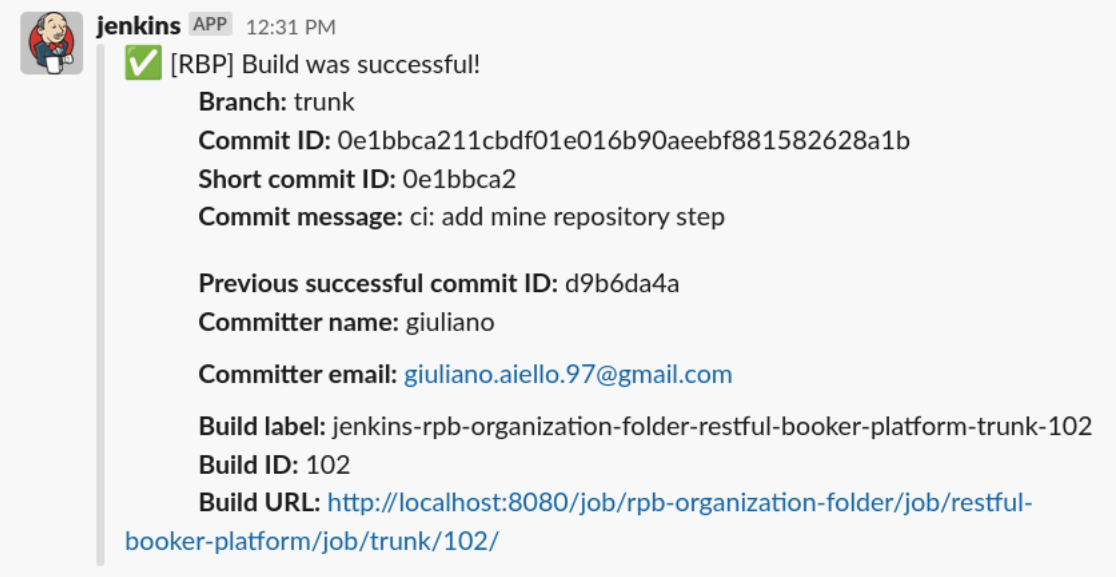


Figura 3.4: Notifica slack personalizzata usando il custom step `rbpSendSlackNotification()`

la notifica contiene informazioni dettagliate sulla build, tra cui il branch corrente, l'ID del commit, il messaggio del commit, l'ID del commit precedente riuscito, il nome e l'email del committer, l'etichetta e l'ID della build, insieme all'URL della build per consentire un rapido accesso ai dettagli sulla Jenkins interface. Questi dettagli forniscono al team una panoramica immediata sullo stato della build, facilitando il processo di tracciamento e gestione delle versioni.

3.2.12.4 Lo `stage` 'Build'

Il primo e reale `stage` che agisce attivamente sul codice di un service è quello in cui tutti i file vengono sottoposti ad un impacchettamento dando vita ad un `.jar`, in sostanza viene effettuata una `build` del modulo, cioè del service corrente. Lo strumento di software project management utilizzato è *Maven*, facendo risultare il corrispondente comando in questo stage.

```
...
stage("[${serviceName}] Build") {
    timeout(time: 2, unit: 'MINUTES') {
        dir("${rbpServiceMainDir}") {
            sh 'mvn clean package -DskipTests'
        }
    }
}
...
```

E' facile dedurre che per eseguire con successo i comandi `mvn`, ogni service ha la propria configurazione *Maven* in un file `pom.xml`, nella rispettiva directory.

3.2.12.5 Gli `stage` 'Unit Tests' e 'Integration Tests'

Il passo successivo della pipeline una volta ottenuta un'entità eseguibile come il file `.jar` del service, è quello di testarla. Utilizzando ancora *Maven*, uno degli strumenti il cui lavoro è preminente lungo tutta l'esecuzione della pipeline, sono state eseguire le istruzioni `mvn test` per gli unit test, e `mvn verify -Dskip.surefire.tests=true` per gli integration test, indicando di saltare i test di unità appena effettuati. Le due tipologie di test sono eseguite in ordine e in due stage differenti, rispettivamente con un tempo massimo –pena fallimento della pipeline– di 20 e 40 secondi.

3.2.12.6 Gli `stage` di static analysis

Necessariamente all'esecuzione di questo stage, è stato utilizzato il tool *SonarQube*, un potente strumento finalizzato all'ispezione e all'analisi statica del codice per il rilevamento di problemi all'interno di esso. Dispone di un vasto insieme di linguaggi di programmazione su cui poter operare.

Per poterlo utilizzare all'interno alla pipeline, come è stato più volte dimostrato, Jenkins permette di effettuare collegamenti con software esterni grazie all'installazione di plugin, anche in questo caso il procedimento è stato lo stesso per permettere l'integrazione di *SonarQube*, e configurarlo nel progetto. Questo ha permesso dunque la scrittura di due `stage` con direttive derivanti dal plugin *SonarQube*: uno per eseguire la scansione del codice sorgente del service, creando così sulla dashboard *SonarQube* un progetto dedicato al service stesso, un altro per permettere alle regole definite di analisi statica di 'intervenire' nella pipeline ed interromperla qualora una di queste lo permettesse, e cioè nel caso in cui una delle regole di scrittura del codice è stata violata.

```
stage("[${serviceName}] SonarQube Scan") {
    ...
```

```
        withSonarQubeEnv(installationName: 'sonarqube') {
            sh """
                mvn sonar:sonar \
                -Dsonar.projectKey=
                    restful-booker-platform-${serviceName} \
                -Dsonar.projectName=
                    restful-booker-platform-${serviceName} \
            """
        }
        ...
    }
    stage("[${serviceName}] Quality Gates") {
        ...
        waitForQualityGate abortPipeline: true
        ...
    }
```

È bene focalizzare l'attenzione sulla gestione delle regole definite sulla piattaforma *SonarQube*: questo tool induce a produrre *Clean Code* attraverso il quality gate predefinito chiamato 'Clean as You Code', rispecchiandosi così nei concetti dell'omonima filosofia. Esso prevede sul **nuovo codice** che il numero di *issue* (un bug, ad esempio) sia pari a 0, che la percentuale di codice coperto dai test automatici sia almeno l'80%, che la percentuale di linee di codice duplicate sia al più del 3%.

3.2.12.7 Lo `stage` 'Build Image'

Lo scopo di questo `stage` è esclusivamente quello di preparare il service alla prossima tipologia di test. Serve in questo caso, un'immagine Docker pronta per essere eseguita dal tool che eseguirà test sulla performance del software.

Questo lo si ottiene con un banale comando `docker build`, alla cui immagine vengono però aggiunte delle etichette per tenere sempre tracciabile e sotto controllo gli artefatti. Ciò si è tradotto quindi in delle opzioni date al comando eseguito:

```
    stage("[${serviceName}] Build Image") {
        ...
        steps {
            sh """
                docker build
                  --build-arg \"BUILD_NUMBER=${BUILD_NUMBER}\" \
                  --build-arg \"BUILD_TAG=${BUILD_TAG}\" \
                  --build-arg \"GIT_COMMIT=${GIT_COMMIT}\" \
```

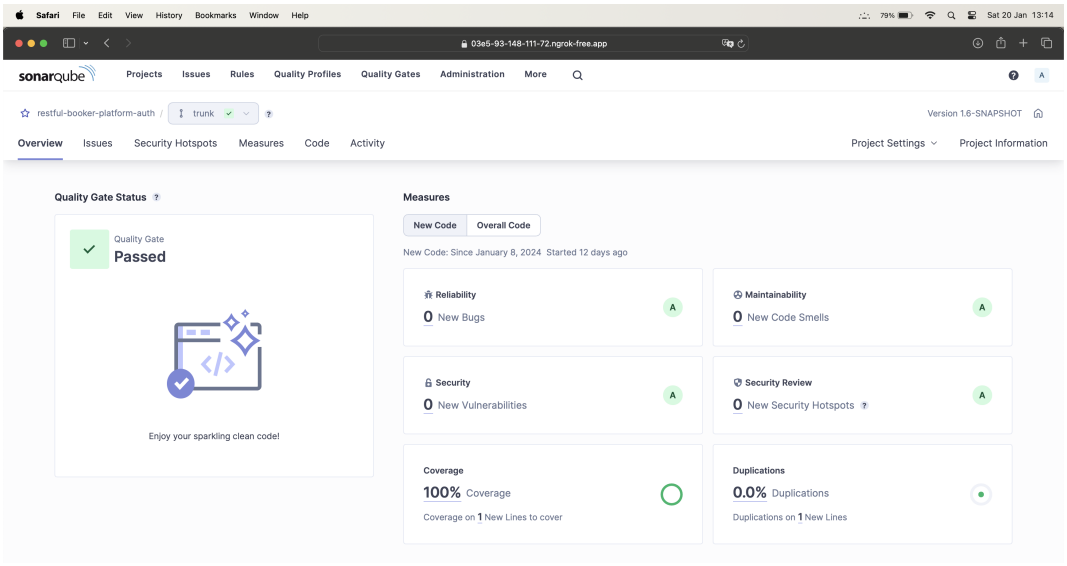


Figura 3.5: Uno screen riportato dall'interfaccia di *SonarQube*

```
                -t \"${DOCKER_REGISTRY_URL}/
                  rbp-${serviceName}:${GIT_SHORT_COMMIT}\" .
            """
        }

        ...
}
```

e in dei parametri di build 'iniettati' nel `Dockerfile`, che viene in parte esposto in seguito.

```
ARG BUILD_NUMBER=0
ARG BUILD_TAG=local
ARG GIT_COMMIT=local

LABEL build_number=${BUILD_NUMBER}
LABEL build_tag=${BUILD_TAG}
LABEL git_commit=${GIT_COMMIT}
```

3.2.12.8 Lo `stage` 'Performance Tests'

Nello stage di "Performance Tests" vengono eseguiti test di carico utilizzando script JMeter specifici per ciascun servizio[2]. Ogni modulo di un servizio contiene una cartella denominata `ci` in cui sono posizionati file, risorse e script eseguibili dedicati alla continuous integration per quel particolare servizio. In questa cartella, è presente uno script JMeter con estensione `.jmx` specifico per il servizio. La cartella deve contenere anche una definizione Docker Compose denominata `docker-compose-test.yaml`, il quale avvia il servizio parametrizzando variabili d'ambiente come il registro Docker da cui prendere le immagini, la porta del servizio e l'immagine Docker del servizio da utilizzare. Prima di eseguire lo script JMeter, è necessario avviare il servizio, eventualmente creando dei mock service per le dipendenze esterne.

L'implementazione dei test delle performance utilizza Taurus, un wrapper che semplifica l'uso di strumenti come JMeter, Gatling, Locust.io e Selenium WebDriver. Taurus consente di definire uno script di performance test in un file YAML. In questo contesto, anziché definire direttamente il test delle performance nel formato YAML, è stato creato uno script JMeter esplicitamente, cui viene fatto riferimento nel file `performance-test.yaml` di Taurus. Per eseguire il test delle performance con Taurus, è necessario installare il tool Taurus (`bzt`) e successivamente eseguire il comando `bzt performance-test.yaml`.

Lo stage di performance test ha l'obiettivo di eseguire load testing, simulando il comportamento degli utenti. I test coinvolgono simulazioni di utenti concorrenti, garantendo che il servizio possa gestire il carico previsto.

I test delle performance non sono sempre semplici da realizzare perché questi richiedono di preparare adeguatamente l'ambiente e di creare eventualmente dei mock per le dipendenze esterne da cui il servizio da testare dipende. Per questo motivo, è stato realizzato soltanto un test delle performance per il servizio `auth`. Nella Shared Library, è stato introdotto un parametro `skipPerformanceTests` che viene settato uguale a `true` per tutti gli altri servizi. Per testare le performance il servizio `auth`, innanzitutto vengono posti degli obiettivi (KPI - Key Performance Indicators). Taurus ci aiuta a definire questi obiettivi tramite uno dei suoi moduli `passfail`. Gli obiettivi da raggiugere sono i seguenti:

- l'endpoint `POST /auth/login` deve avere un tempo di risposta medio minore di 100ms;
- l'endpoint `POST /auth/validate` deve avere un tempo di risposta medio minore di 80ms;

[2]JMeter è una applicazione Java progettata per eseguire test di carico e misurare le performance.

- l'endpoint `POST /auth/logout` abbia un tempo di risposta medio di 100ms;
- il tempo di risposta medio complessivo deve essere minore di 100ms;
- tutte le richieste durante l'esecuzione dei test delle performance devono restituire un HTTP status code positivo.

Tramite Taurus questi obiettivi possono essere facilmente settati in questo modo:

```
- module: passfail
  criteria:
      - avg-rt of LoginPostEndpoint>=100ms for 5s, stop as failed
      - avg-rt of ValidatePostEndpoint>=80ms for 5s, stop as failed
      - avg-rt of LogoutPostEndpoint>=100ms for 5s, stop as failed
      - avg-rt >=100ms for 5s, stop as failed
      - fail >0 for 1s, stop as failed
...
```

Lo script JMeter simula un utente in questo modo:

1. l'utente esegue il login una volta;
2. l'utente valida il token 3 volte con una pausa di 2-4 secondi tra una chiamata e l'altra;
3. l'utente esegue il logout.

Questo scenario viene chiamato `login-validate-logout`. Vengono eseguiti 20 utenti concorrenti con un ramp-up di 5 secondi. Lo script JMeter ripete la stessa procedura per 3 volte prima di terminare.

Questo scenario può essere "formalizzato" con Taurus in YAML nel seguente modo:

```
---
settings:
  env:
    CSV_TEST_USERS_DATA_PATH: ./postgres/test-users.csv
    JMETER_HOME: ~/apache-jmeter-5.6.2/bin
    RBP_SERVICE_HOSTNAME: localhost
    RBP_SERVICE_PORT: 3004

execution:
  - concurrency: 20
    iterations: 3
    ramp-up: 5s
```

```
    scenario: login-validate-logout

scenarios:
  login-validate-logout:
    script: performance-test-jmeter.jmx
    variables:
      CSV_TEST_USERS_DATA_PATH: ${CSV_TEST_USERS_DATA_PATH}
      RBP_AUTH_SERVICE_HOSTNAME: ${RBP_SERVICE_HOSTNAME}
      RBP_AUTH_SERVICE_PORT: ${RBP_SERVICE_PORT}
...
```

Nella Figura 3.6 vengono mostrati gli elementi creati sulla interfaccia grafica di JMeter affinché lo script possa essere eseguito come appena descritto.

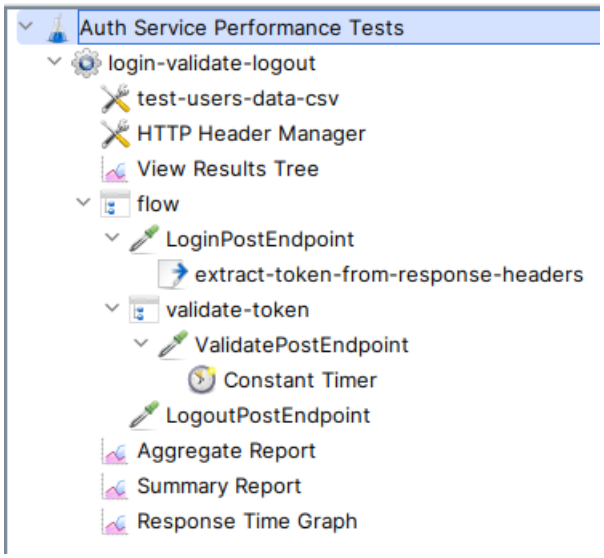


Figura 3.6: Lo script JMeter per il servizio `auth`

I risultati dei test delle performance vengono generati usando due moduli di Taurus: `final-stats` e `junit-xml` (per generare i report in un formato xml stile JUnit).

```
...
reporting:
  - module: final-stats
    summary: true  # overall samples count and percent of failures
    percentiles: true  # display average times and percentiles
    summary-labels: true # provides list of sample labels, status, percentage of
```

```
    failed-labels: true  # provides list of sample labels with failures
    test-duration: true  # provides test duration
    dump-xml: rbp-auth-performance-report.xml
  - module: junit-xml
    filename: rbp-auth-junit-performance-report.xml
    data-source: pass-fail
  ...
```

L'ultimo modulo (`junit-xml`) permette a Jenkins di leggere questi report e mostrare dei grafici sull'interfaccia grafica del Jenkins controller. La Figura 3.7 mostra i grafici appena menzionati.

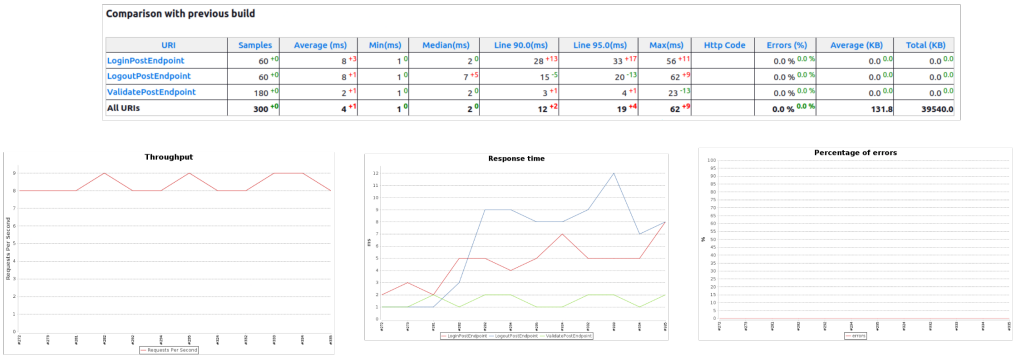


Figura 3.7: Report dei test delle performance del servizio `auth`

Di seguito viene riportato lo `stage` di "Performance Tests" implementato nella Shared Library.

```
...

stage("[${serviceName}] Performance Tests") {
    if (skipPerformanceTests == true) {
        echo "[${serviceName}] Skip Performance Tests Stage"
    } else {
        env.RBP_SERVICE_DOCKER_IMAGE_TAG = "${GIT_SHORT_COMMIT}"
        env.RBP_SERVICE_HOSTNAME = "${rbpServiceHostname}"
        env.RBP_SERVICE_PORT = "${rbpServicePort}"

        timeout(time: 1, unit: 'MINUTES') {
            dir("${rbpServiceCiDir}") {
                sh '''
                 docker compose \
```

```
                    -f docker-compose-test.yaml \
                    up -d --build --wait
                '''
                bzt """
                 -o settings.env.JMETER_HOME=${JMETER_HOME} \
                 -o settings.env.RBP_SERVICE_HOSTNAME=${rbpServiceHostname} \
                 -o settings.env.RBP_SERVICE_PORT=${rbpServicePort} \
                 performance-test.yaml
                """
            }
        }
    }
}

...
```

3.2.12.9 Lo `stage` 'Push Image'

Qui si affronta l'ultimo **stage** appartenente alla libreria dopo che il service ha appena superato con successo più tecniche di testing. Avendo a disposizione la docker image dello stato corrente del service, questa viene ora pubblicata su un registry privato. La motivazione di quest'operazione, seppur identica ad una presunta fase di *delivery*, non viene considerata come tale, lo scopo infatti è quello di disporre il service ad un'altra –ed ultima– tipologia di test: l'End-to-End testing (dettagliato in 3.2.13).

```
stage("[${serviceName}] Push Image") {
  if (currentBuild.result != null && currentBuild.result != 'SUCCESS') {
        echo "[${serviceName}] Pipeline not in a
          successful state, skipping Push Image stage..."
    } else {
        def dockerImageBaseTag =
            "${DOCKER_REGISTRY_URL}/rbp-${serviceName}"
        def commitDockerImage =
            dockerImageBaseTag + ":${GIT_SHORT_COMMIT}"
        def latestDockerImage =
            dockerImageBaseTag + ":latest"
        timeout(time: 1, unit: 'MINUTES') {
            sh """
                docker tag ${commitDockerImage} ${latestDockerImage} && \
                docker push ${commitDockerImage} && \
```

```
                    docker push ${latestDockerImage}
                """
            }
        }
    }
```

Possiamo notare che viene considerato il caso in cui lo stato attuale della pipeline non stia andando a buon fine, chiaramente questo comporta una successiva attenta analisi da parte degli sviluppatori, avendo a disposizione gli elementi "incriminati": l'ultimo commit, la docker image, e il file `.jar`, come vedremo.

3.2.12.10 Il blocco `finally` - raccolta dei report

A questo punto, la pipeline ha usato molti strumenti per mettere 'sotto pressione' il service, questi però possono chiaramente portare ad un fallimento della stessa, precisamente in una modalità *"fail-fast"* come visto in 3.2.6. Ed è proprio il blocco `finally` che un service affronterà come ultimo passo della shared library, indipendentemente che lo stato della pipeline sia con successo o meno. Vengono di seguito elencate alcune delle operazioni svolte in questo blocco.

- `junit`, in questa direttiva, permessa dall'omonimo plugin installato sul Jenkins controller, vengono presi in input i report generati dai test automatici effettuati con *Maven*, in particolare gli unit test, gli integration test e i performance tests. Così facendo, le informazioni dei test sono disponibili per la consultazione proprio sul Jenkins controller.

- `jacoco`, anche qui, avendo installato il plugin sul Jenkins controller, sono state abilitate delle direttive per il tool *JaCoCo*, software per l'analisi statica, questo è utile perché offre altre prospettive sul codice come i report sulla copertura da parte dei test automatici sulle singole istruzioni, sui singoli metodi e sulle singole classi.

- `archiveArtifacts`, con questa direttiva è stato possibile archiviare artefatti del service; in questo caso è stato archiviato il file `.jar` corrispondente al microservizio, che sarà dunque disponibile sul Jenkins controller per un'analisi sia in caso di successo della pipeline, che di fallimento. Questo è il risultato di ciò che suggerisce anche la filosofia di DevOps, in quanto gli artefatti debbano essere tracciati.

- `recordIssues`, grazie al plugin 'Warnings Next Generation', questa direttiva permette di avere una visione complessiva in un'unica interfaccia grafica sui problemi che il codice può aver generato. In questo caso è stata presa la scelta di aggregare insieme gli warnings dati eventualmente dagli unit test e dagli integration test.

3.2.13 Lo `stage` 'E2E Tests'

Una volta terminate le pipeline individuali di ogni servizio, il prossimo `stage` da eseguire è quello dei test end-to-end. I test E2E sono stati implementati usando Selenium WebDriver per automatizzare le operazioni da eseguire su un determinato browser. Lo `stage` E2E è pensato per eseguire in parallelo gli stessi test E2E su almeno tre differenti browser: Chrome, Firefox ed Edge. Per eseguire in parallelo i test usiamo la direttiva `parallel` già presentata in Sezione 3.2.10. La struttura dello `stage` nel `Jenkinsfile` di RBP è la seguente:

```
...

stage('E2E Tests') {
    environment {
        RBP_TEST_PILOT_MAIN_DIR = 'test-pilot'
        RBP_TEST_PILOT_DOCKER_DIR =
            "${RBP_TEST_PILOT_MAIN_DIR}/src/test/resources/docker"
        DISABLE_TESTCONTAINERS = 'true'
        RBP_PROXY_URL = 'http://rbp-proxy:8080'
    }

    options {
        timeout(time: 3, unit: 'MINUTES')
    }

    parallel {
        // Stage 'Chrome': esegui test E2E su Chrome
        // Stage 'Firefox': esegui test E2E su Firefox
        // Stage 'Edge': esegui test E2E su Edge
    }
}

...
```

Nel blocco di codice fornito, sono definite diverse variabili d'ambiente utilizzate per lo stage 'E2E Tests'. In particolare, le variabili `RBP_TEST_PILOT_MAIN_DIR` e `RBP_TEST_PILOT_DOCKER_DIR` puntano rispettivamente alla directory principale del modulo `test-pilot` e alla sottodirectory `docker` all'interno di questo modulo, che contiene la definizione `docker-compose-test.yaml` necessaria per eseguire l'intera piattaforma prima dell'esecuzione dei test. La variabile `DISABLE_TEST_CONTAINERS` è utilizzata per disabilitare l'uso della libreria Testcontainers all'interno di `test-pilot`,

la quale era inizialmente progettata per avviare automaticamente la piattaforma RBP per i test E2E in un ambiente locale sulla macchina dello sviluppatore. Questa libreria è stata disabilitata per consentire l'esecuzione parallela dei test E2E. Invece di affidarsi automaticamente a Testcontainers, il file `docker-compose-test.yaml` viene eseguito manualmente all'interno dei rispettivi `stage` di 'Chrome', 'Firefox' e 'Edge', garantendo maggiore controllo sulla parallelizzazione dei test.

È sempre buona pratica settare un `timeout` come già accennato nei precedenti paragrafi. In questo caso, il `timeout` è impostato a 3 minuti.

Affinché possano essere avviati in parallelo i test E2E sui differenti browser, è necessario configurare i `WebDriver` corrispondenti in ogni `stage`. Per fare ciò, è stata definita una variabile d'ambiente `WEB_DRIVER` il cui valore determina l'istanza di `WebDriver` da creare. In particolare:

- `WEB_DRIVER='chrome'`: istanzia un oggetto `ChromeDriver()`;
- `WEB_DRIVER='edge'`: istanzia un oggetto `EdgeDriver()`;
- `WEB_DRIVER='firefox'`: istanzia un oggetto `FirefoxDriver()`.

Ciascuno di questi `stage` ('Chrome', 'Firefox' ed 'Edge') esegue su un nuovo docker container. Ogni container punta allo stesso Docker engine. Per questo motivo, per evitare conflitti, bisogna specificare una rete Docker e un Compose project name differente. Ogni `stage`, oltre alla variabile `WEB_DRIVER`, deve definire quindi altre due variabili il cui valore deve essere unico:

- `RBP_E2E_DOCKER_NETWORK`: questa variabile definisce il nome della rete docker che deve essere usata. Ogni `stage` deve avere una propria rete docker isolata affinché i test possano essere eseguiti in isolamento;
- `RBP_E2E_DOCKER_PROJECT_NAME`: questa variabile definisce il nome del progetto Compose compose da usare. Il valore deve essere unico. Il motivo di ciò è dovuto dal fatto che i tre `stage` ('Chrome', 'Firefox' ed 'Edge') avviano lo stesso `docker-compose-test.yaml` e puntano allo stesso Docker engine.

Le reti Docker sono state create manualmente con i seguenti nomi:

- `RBP_E2E_DOCKER_NETWORK='rbp-e2e-chrome'` per lo `stage` 'Chrome';
- `RBP_E2E_DOCKER_NETWORK='rbp-e2e-edge'` per lo `stage` 'Edge';
- `RBP_E2E_DOCKER_NETWORK='rbp-e2e-firefox'` per lo `stage` 'Firefox'.

La variabile `RBP_E2E_DOCKER_NETWORK` una volta settata sarà riconosciuta e iniettata automaticamente da Docker Compose quando si esegue il comando `docker compose up` del file `docker-compose-test.yaml`. Il valore della variabile `RBP_E2E_DOCKER_PROJECT_NAME` viene settato in maniera analoga con lo stesso identificativo.

Di seguito viene riportato lo `stage` 'Chrome'. Gli `stage` 'Edge' e 'Firefox' sono definiti in maniera analoga e quindi omessi:

```
...

parallel {
    stage('Chrome') {
        environment {
            WEB_DRIVER = 'chrome'
            RBP_E2E_DOCKER_NETWORK = 'rbp-e2e-chrome'
            RBP_E2E_DOCKER_PROJECT_NAME = 'rbp-e2e-chrome'
        }

        agent {
            label 'rbp-e2e-chrome'
        }

        steps {
            unstash 'rbp'

            dir("${RBP_TEST_PILOT_DOCKER_DIR}") {
                sh """
                    docker compose -f docker-compose-test.yaml \
                        -p ${RBP_E2E_DOCKER_PROJECT_NAME} \
                        up -d
                """
            }

            dir("${RBP_TEST_PILOT_MAIN_DIR}") {
                sh 'mvn clean test -Dcucumber.features=src/test/resources'
            }
        }

        post {
            always {
                dir("${RBP_TEST_PILOT_DOCKER_DIR}") {
```

```
                    sh """
                        docker compose -f docker-compose-test.yaml \
                            -p ${RBP_E2E_DOCKER_PROJECT_NAME} \
                            down || :
                    """
                }

                dir("${RBP_TEST_PILOT_MAIN_DIR}") {
                    junit(
                        testResults: 'target/surefire-reports/**/*.xml',
                        allowEmptyResults: true
                    )
                }
            }
        }
    }

    stage('Firefox') {
        ...
    }

    stage('Edge') {
        ...
    }
}

...
```

Da come si può notare, ogni `stage` definisce anche il proprio `agent`. Ognuno di questi `agent` ha la stessa `label` della rete Docker e del Compose project name. Gli `steps` eseguiti sono:

- Unstash della repository Git con `unstash 'rbp'` (vedere Sezione 3.2.9);
- Esecuzione di tutti i servizi con Docker Compose up;
- Esecuzione dei test E2E usando Cucumber[3];
- Alla fine, in ogni caso, elimina i container avviati e registra i risultati dei test E2E.

[3]Cucumber è uno strumento per eseguire test automatizzati scritti secondo lo stile di sviluppo basato sul comportamento (Behavior-driven Development - BDD)

La Figura 3.8 riassume a grandi linee l'intero `stage` 'E2E Tests'.

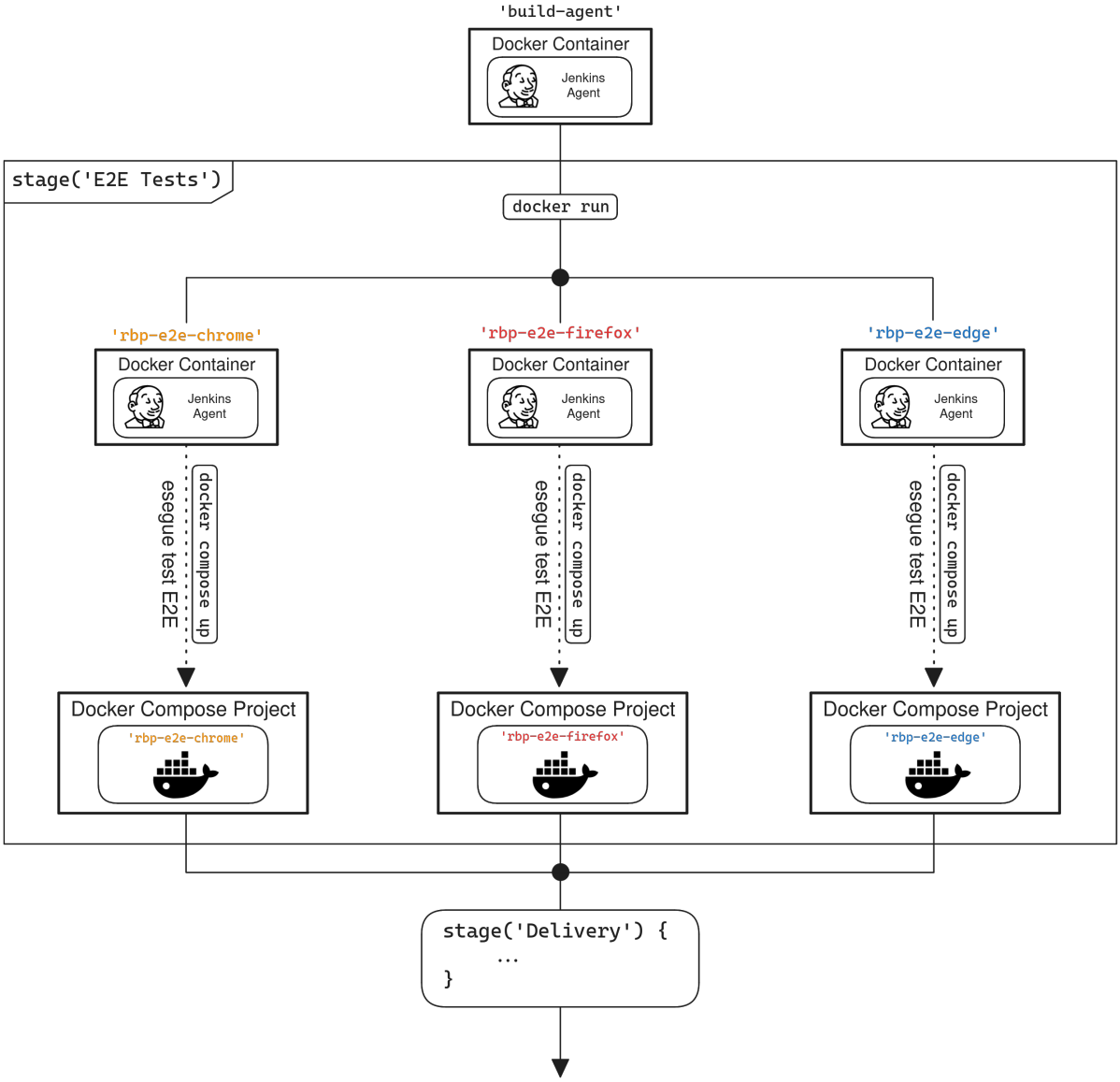


Figura 3.8: Lo `stage` 'E2E Tests'

3.2.14 Lo `stage` 'Delivery'

Nello stage 'Delivery', giungiamo al culmine della pipeline CI/CD, dove tutti gli `stage` precedenti, sia a livello di singoli servizi che di integrazione dell'intera piattaforma, hanno superato con successo i rispettivi test. Praticamente tutte le pipeline individuali dei servizi coinvolti nella modifica hanno dato esito positivo, evidenziando una solidità a livello microservizio. Inoltre, l'intero sistema è stato sottoposto a test end-to-end sofisticati, confermando la sua resilienza e coerenza nell'ambito di scenari complessi e realistici.

Giunti a questo punto, è il momento di consolidare il successo raggiunto e prepararsi per il rilascio (non in produzione). Il passo fondalmente è l'invio delle immagini Docker generate durante la pipeline in un registro privato, garantendo così un'archiviazione sicura e accessibile. Prima di effettuare il push delle immagini, viene attuata una "pratica di certificazione": una `label`, denominata 'e2e_test_passed', viene assegnata all'immagine Docker e impostata su 'true'. Questo marcaggio certifica che l'immagine ha superato con successo i test end-to-end, fornendo una chiara indicazione del suo stato di integrità e qualità. Solo dopo questa certificazione, l'immagine Docker viene pushata nel registro privato, consolidando così il successo della pipeline e preparando il terreno per una possibile distribuzione del software in produzione.

Per eseguire questo passaggio, è stato realizzato uno script bash apposito. Lo script si chiama `delivery_image` e può essere trovato alla root della repository `restful-booker-platform`.

```
    ...

    stage('Delivery') {
       environment {
           CI_DIR = 'ci'
       }


       steps {
           dir("${CI_DIR}") {
               sh './image_delivery.sh'
           }
       }
    }

    ...
}
```

Conclusioni

In conclusione, affrontare questa sfida ci ha fornito un'opportunità preziosa per esplorare in profondità il processo di progettazione e implementazione di una robusta pipeline di Continuous Integration e Continuous Delivery (CI/CD). Grazie all'utilizzo avanzato di Jenkins come strumento chiave per l'automazione, abbiamo acquisito una comprensione approfondita delle pratiche e delle metodologie di CI/CD, consolidando competenze significative in un ambito cruciale per lo sviluppo software moderno.

La realizzazione pratica della pipeline ha richiesto l'applicazione di concetti avanzati, come la gestione efficace delle dipendenze, la parallelizzazione dei test, e la "certificazione" delle immagini Docker attraverso l'etichettatura.

La scelta di Jenkins come automation server ha dimostrato di essere fondamentale, offrendo una vasta gamma di funzionalità e una flessibilità che si è adattata in modo impeccabile alle esigenze del progetto. Quest'esperienza ci ha consentito di padroneggiare le peculiarità e le sfide dell'implementazione di CI/CD in un contesto complesso, preparandoci a gestire progetti futuri in modo più efficiente e resiliente.

La progettazione e l'implementazione di una pipeline di CI/CD non solo ha arricchito la nostra competenza tecnica, ma ha anche consolidato la consapevolezza dell'importanza cruciale di una pipeline CI/CD ben progettata e gestita nel contesto dello sviluppo software contemporaneo. Questa esperienza ci fornirà una solida base per affrontare sfide più ampie e complesse nel nostro percorso professionale.

Il progetto è disponibile al seguente link GitHub https://github.com/vtramo/restful-booker-platform.