UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato Finale In Ingegneria Del Software

Automazione Del Testing e Della Valutazione Della Qualità Con Github Actions e YAML

Anno Accademico 2021/2022

Candidato
Alex Lakshan Fernando Kurukulasuriya
matr. N46003827

ALLE CADUTE

Indice

Introduzione

0.1 Il software e le fasi di sviluppo

Il Software è un insieme di componenti, procedure e istruzioni che compongono un sistema di elaborazione dati. Il programmatore è un tecnico che, attraverso la programmazione, crea le funzionalità richieste. In pratica, scrive il codice che serve per far funzionare una o più parti del software. Oltre a sviluppare il codice necessario, può apportare un contributo importante alla visione d'insieme del progetto. Oltre al programmatore ci sono altre figure importanti che contribuiscono allo sviluppo di un software, come, *Project manager, Web designer, Systems architect o software architect* ecc.. Possiamo distinguere i software in base a chi li utilizzerà:

- **Software di Sistema :** Sono i software progettati per fornire supporto ad un altro software: I sistemi operativi (OS), come macOS, Android, Microsoft Windows e GNU/Linux sono tutti esempi di software di sistema.
- **Software applicativi :** Sono i programmi applicativi che ci aiutano a svolgere determinate funzioni

Quando si parla di ciclo di vita del software, o anche *Systems Development Life Cycle*, ci si riferisce all'insieme delle fasi e delle attività necessarie per realizzare il software. Può anche variare in base alle caratteristiche del software e alle necessità del progetto, ma di solito le fasi del ciclo di vita sono:

- **Analisi :** L'indagine preliminare che serve a definire il contesto, le caratteristiche, le funzionalità principali del software. Si cerca di capire e documentare ciò che è richiesto dagli utenti e da altre parti interessate.
- **Progettazione :** In questa fase si entra nel dettaglio di come la soluzione deve essere costruita, definendo architettura e struttura dei singoli componenti.
- **Programmazione o Implementazione :** Costruzione di un codice nel linguaggio di programmazione appropriato per la costruzione del vero e proprio software.

- **Testing :** Per verificare che i requisiti definiti in fase di analisi vengano rispettati e che non si verifichino errori mediante alcuni scenari pre-pianificati come parte della progettazione e della codifica del software.
- **Deployment :** Il prodotto è pronto per essere rilasciato. Questa fase può prevedere anche l'installazione e la configurazione del software sulle macchine che compongono l'infrastruttura tecnologica che verrà utilizzata dall'utente finale. Il software può essere rilasciato anche in un ambiente di testing per verificare che non ci siano errori.
- **Manutenzione :** Gli utenti hanno iniziato ad utilizzare il prodotto rilasciato e lo sviluppatore si occupa di fare gli aggiustamenti necessari a migliorare il software. La manutenzione può essere:
  - Manutenzione correttiva
  - Manutenzione adattiva
  - Manutenzione evolutiva

0.2 Il software testing automatizzato

Come qualunque altro prodotto sul mercato, anche un programma software non è mai perfetto: richiede una fase di testing prima di essere distribuito, commercializzato o messo in produzione all'interno di un'organizzazione. La fase di testing dunque ha come obiettivo quello di elevarne la qualità e verificare che le richieste dell'utente siano quanto più soddisfatte. Il superamento di tutti i test però non garantisce che il software, una volta rilasciato, funzionerà come dovrebbe.

La fase di testing viene spesso eseguita più avanti nel ciclo di vita dello sviluppo del software dopo la fase di creazione o esecuzione del prodotto. Un *tester* può avere solo una piccola finestra per testare il codice, a volte appena prima che l'applicazione venga immessa sul mercato. Se vengono rilevati difetti, potrebbe essere necessario la ricodifica o il nuovo test. Non è raro rilasciare software in tempo, ma con bug e correzioni necessarie. Oppure un team di test potrebbe impegnarsi a correggere gli eventuali errori senza però riuscire a rispettare una data di rilascio. Potrebbe essere vantaggioso mantenere la fase di testing all'inizio del ciclo di sviluppo per rilevare man mano gli errori con lo scopo di prevenire i difetti troppo costosi da risolvere.

Il testing del software è quindi molto importante per garantire la qualità ed una *user experience* migliore all'utente, sia che venga eseguito manualmente o attraverso metodi automatici. Si valuta se le funzioni richieste non presentino difetti, ad esempio , se

sono conformi alle normative di un determinato settore. In altri termini si effettua una ***Verification*** (il prodotto creato è corretto) e una ***Validation*** (è stato creato il prodotto corretto).
Si possono distinguere due principali approcci di analisi di test, entrambi automatizzabili: Analisi statica, che esamina il codice del programma e la documentazione ad esso associata e Analisi dinamica che valuta il comportamento del codice mentre è in esecuzione.
Oggigiorno qualsiasi tipo di software viene realizzato in team, questo significa che c'è bisogno di una costante comunicazione tra i componenti del team ogni volta che il software viene modificato. Un concetto importante in questo caso è quello di ***Regression testing*** che consiste nella ri-esecuzione , totale o parziale, dei test precedentemente scritti, per assicurarsi che i cambiamenti più recenti non abbiano introdotto nuovi bugs (se ciò non dovesse risultare verificata allora siamo in presenza di una regressione).

A causa della forte concorrenza tra le aziende, quest'ultime stanno adottando delle tecniche agevoli per automatizzare i test di software per ottenere versioni più veloci e prodotti di qualità. Test manuali o test ad hoc possono essere sufficienti per piccole build. Tuttavia, per i sistemi più grandi, gli strumenti vengono spesso utilizzati per automatizzare le attività. I test automatizzati aiutano i team a implementare diversi scenari, testare fattori di differenziazione (come lo spostamento di componenti in un ambiente *cloud*) e ottenere rapidamente *feedback* su cosa funziona e cosa no.

Dunque *Automation Testing* è una tecnica di testing del software che viene eseguita utilizzando speciali strumenti software di test automatizzati per eseguire una *suite* di *test case*. I cicli di sviluppo successivi richiederanno l'esecuzione ripetuta della stessa suite di test. Utilizzando uno strumento di automazione dei test, è possibile registrare questa suite di test e riprodurla come richiesto. Una volta che la suite di test è automatizzata, non è richiesto alcun intervento umano. È il modo migliore per aumentare l'efficacia, la copertura dei test e la velocità di esecuzione nei test del software.

0.3 Vantaggi dell'automazione del testing

L'utilizzo dei *tools* per eseguire test comporta potenziali vantaggi ed opportunità:

- Aumenta la copertura e l'ambito dei test per migliorare la qualità delle applicazioni. Aiuta anche a risparmiare tempo e riduce notevolmente lo sforzo manuale.
- Poichè i cicli di test manuali potrebbero portare a errori, con strumenti di test automatizzati, vi è garanzia di una buona precisione.

- Facilità gli sforzi del *tester* poichè gli *script* di test possono essere riutilizzati con alcune piccole modifiche.
- L'esecuzione dei test manuali richiede molto tempo e sforzi umani che possono anche portare a molti *bug* non identificati. Dunque l'utilizzo di test automatizzati risolve questo problema con vari strumenti di test ed i tester manuali ottengono più tempo per essere utili per altri lavori di qualità.

Nonostante i numerosi vantaggi che possiamo ottenere con questo tipo di testing, si presenta la necessità di mantenere una comunicazione efficace tra i team per garantire rilasci più rapidi e il successo dell'automazione dei test. Quindi è necessario, a tale scopo , scrivere degli opportuni script di test, ovvero un insieme di istruzioni che vengono eseguiti sul sistema, per verificarne il corretto comportamento. La corretta scrittura di questi test consente di risparmiare sia il tempo che le risorse, dando la possibilità allo sviluppatore di risolvere il problema, o perlomeno di identificare la fonte di provenienza.

Come vedremo nei capitoli più avanti, è possibile utilizzare degli appositi strumenti per effettuare in maniera efficiente il procedimento di testing automatizzato (Dunque è essenziale scegliere lo strumento giusto) in base all'applicazione sottoposta ai test (applicazioni web, i dispositivi mobili ecc) per ottenere un software di qualità. Inoltre introdurremo un esempio completo di *Automation Testing* utilizzando un progetto Java e vari strumenti per la creazione, condivisione ed esecuzione di test suite.

Capitolo 1

Strumenti utilizzati

1.1 GitHub

I principali concetti su cui si basa il *GitHub* possono essere: rendere il software sempre più perfetto e di creare delle porzioni di software "riutilizzabili" per più scopi. Non é altro che un servizio di *Hosting* su cui lo sviluppatore, dopo essersi registrato, puó caricare il proprio software, rendendolo aperto a chiunque vorrá contribuire allo sviluppo, oppure a chiunque vorrá utilizzarlo nel proprio software. Ad esempio, ci si collega su GitHub, si cerca il progetto piú adatto alle proprie esigenze e che sia scritto con il linguaggio interessato, si fa una valutazione sulla base di alcune informazioni e lo si integra.

Il protocollo **git** è particolarmente utilizzato in progetti *open source* distributi sul *Web*, per la gestione delle versioni. GitHub, invece, é un servizio di hosting e di *team collaboration*, basato sulle funzionalità di Git. Vediamo alcuni concetti importanti del GitHub,

- **Il Repository :** Abbiamo bisogno di iniziare a condividere il codice sorgente del nostro progetto. Il primo *step* é la creazione di un contenitore dove andare a salvare il nostro codice, nelle versioni *master* (ovvero, di produzione), *develop* (di sviluppo) ed altre. Questo contenitore, appunto, si chiama *Repository*. In altre parole, nasce concettualmente come un pozzo di *storage* di tutta la storia del codice e degli altri artefatti di un progetto, durante tutto il suo ciclo di vita. Ogni contributore del progetto, quindi, avrá a disposizione in locale e in remoto, una copia del Repository con tutto il codice sorgente e la documentazione create dagli altri contributori per poi andare a modificare eventualmente. Vedi figura 1.1:

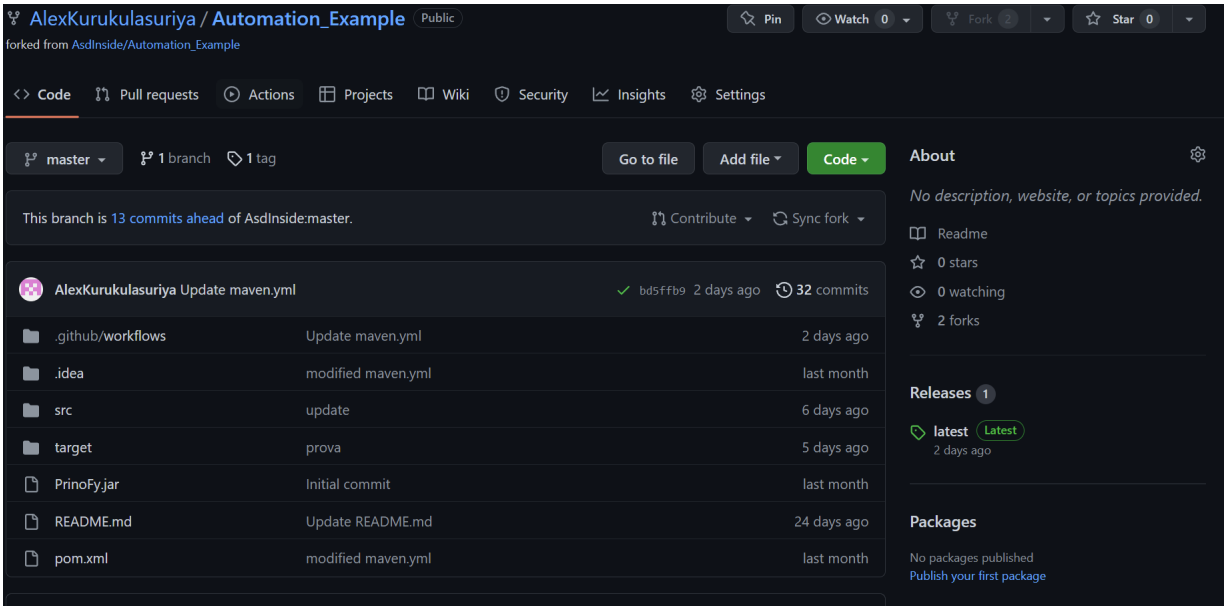

Figura 1.1: Esempio di un Repository

- **Il Branch :** Un *branch* é una sottoparte del Repository che include il codice di una certa versione, derivato direttamente da un altro branch o dalla fusione (*merge*) di piú branch differenti. Per non creare caos e stabilire quale elenco di funzionalitá debba contenere una certa versione del software, Git (e GitHub), implementano il concetto di Branch.

- **Il Commit, Push e Pull :**

  - Quando i contributori iniziano a scrivere il codice in modo organizzato, lo faranno in locale, ognuno sul loro *PC*. Per inviare le proprie modifiche al branch sará necessario eseguire un ***commit***. Possiamo definire, quindi, il commit come l'azione con cui si decide di trasferire il cambiamento apportato al codice, rendendolo disponibile a chiunque abbia accesso al Repository.
  - Con l'azione di commit, in realtà, non si invia nulla al Repository remoto, ma si genera solo un pacchetto contenente le modifiche. Per inviare il pacchetto bisogna usare il comando ***Push***, mediante il quale il Repository acquisisce le modifiche da un contributore, rendendole disponibili agli altri.
  - Ogni contributore, per acquisire nel loro Repository locale le modifiche inviate dagli altri, deve effettuare il comando di ***Pull***, mediante il quale il Repository locale viene allineato con le modifiche presenti sul remoto.

1.2 GitHub Actions

Le azioni di GitHub Actions consentono agli sviluppatori di creare *workflow* ***SDLC*** *(Software Development Life Cycle)* automatizzati. È sufficiente un *Repository GitHub* per creare ed eseguire un *workflow GitHub Actions*. Si tratta di una piattaforma di integrazione continua e distribuzione continua **(CI/CD)** che consente di automatizzare la *pipeline* di compilazione, test e distribuzione. Ci consente di creare i workflow che compilano e testano ogni richiesta pull nel repository o distribuire richieste pull unite alla produzione. Inoltre è possibile eseguire un workflow per aggiungere automaticamente le etichette appropriate ogni volta che qualcuno crea un nuovo problema nel Repository. GitHub Actions fornisce macchine virtuali Linux, Windows e macOS per eseguire i workflow oppure è possibile ospitare i propri corridori (*Runners*) *self-hosted* nel proprio *data center* o infrastruttura cloud.

È possibile configurare un workflow GitHub Actions da attivare quando si verifica un evento nel Repository. Il workflow contiene uno o più processi che possono essere eseguiti in ordine sequenziale o in parallelo. Ogni processo verrà eseguito all'interno del proprio canale macchina virtuale o all'interno di un contenitore e prevede uno o più passaggi che eseguono uno script definito o un'azione.

1.2.1 Componenti Principali

- **Workflow :** Un flusso di lavoro (*workflow*) è un processo automatizzato configurabile che eseguirà uno o più processi. I workflow sono definiti mediante un file ***YAML*** archiviato nel Repository e verranno eseguiti quando vengono attivati da un evento nel Repository oppure possono essere attivati manualmente o in base ad una pianificazione definita. Un Repository può avere più flussi di lavoro, ognuno dei quali può eseguire un diverso set di attività. Un workflow deve contenere i seguenti componenti di base:

  - Uno o più eventi che attiveranno il flusso di lavoro.
  - Uno o più *jobs*, ognuno dei quali verrà eseguito su una macchina runner ed eseguirà una serie di passaggi .
  - Ogni passaggio può eseguire uno script definito dall'utente o eseguire un'azione, che è un'estensione riutilizzabile che può semplificare il flusso di lavoro.

  Come illustrato nella figura seguente,

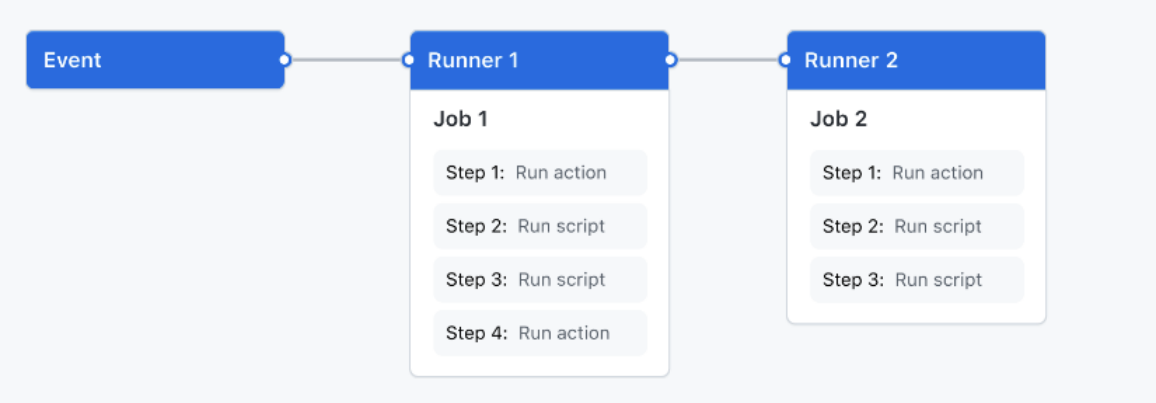

Figura 1.2: Struttura di un Workflow

- **Evento :** Un evento è un'attività specifica in un Repository che attiva l'esecuzione di un workflow tramite una richiesta pull, creazione di un problema o invio un commit a un Repository (Ad esempio, se non viene specificato alcun tipo di attività, il workflow viene eseguito quando viene aperta o riaperta una richiesta pull o quando viene aggiornato il ramo *head* della richiesta pull).
- **Jobs :** Un Job è un insieme di passaggi in un workflow che vengono eseguiti sullo stesso corridore. Ogni Job è uno script di *shell* che verrà eseguito o un'azione che verrà eseguita. I passaggi vengono eseguiti in ordine e dipendono l'uno dall'altro. Poiché ogni passaggio viene eseguito sullo stesso corridore, è possibile condividere i dati da un passaggio all'altro. I jobs che non hanno dipendenze vengono eseguiti in parallelo tra loro. Quando un processo assume una dipendenza da un altro processo, attenderà il completamento del processo dipendente prima di poter essere eseguito. Ogni Job viene eseguito in un ambiente corridore specificato da **runs-on**. Si può eseguire un numero illimitato di jobs purché rientri nei limiti di utilizzo del workflow. Nel seguente esempio (fig. 1.3) sono stati creati due Jobs e i loro **job-id** valori sono **my-first-job** e **my-second-job**:

```yaml
jobs:
  my_first_job:
    name: My first job
  my_second_job:
    name: My second job
```

Figura 1.3: Esempio Jobs

- **Actions :** Un'azione è un'applicazione personalizzata per la piattaforma GitHub Actions che esegue un'attività complessa ma ripetuta frequentemente. È possibile

utilizzare un'azione per ridurre la quantità di codice ripetitivo scritto nei file del workflow.

- **Corridori(Runers) :** Un corridore è un *server* che esegue i workflow quando vengono attivati. Ogni corridore può eseguire un singolo job alla volta. GitHub fornisce i corridori Ubuntu Linux, Microsoft Windows e macOS per eseguire i workflow, per esempio,

```
runs-on: ubuntu-latest
```

Figura 1.4: Sintassi per specificare il Runner

ogni esecuzione del workflow viene eseguita in una nuova macchina virtuale di cui è stato eseguito il *provisioning*. La macchina virtuale contiene un ambiente di strumenti, pacchetti e impostazioni disponibili per l'utilizzo di GitHub Actions. Se abbiamo bisogno di un sistema operativo diverso o di una configurazione hardware specifica, possiamo ospitare i nostri corridori, per esempio,

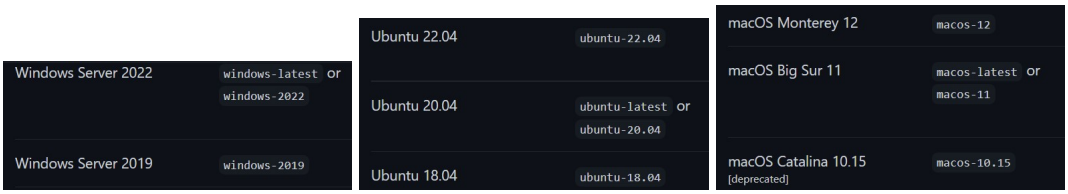

Figura 1.5: Vari Runners disponibili

Nel diagramma seguente viene illustrato come vengono eseguiti due processi in un flusso di lavoro in due diversi corridori ospitati da GitHub.

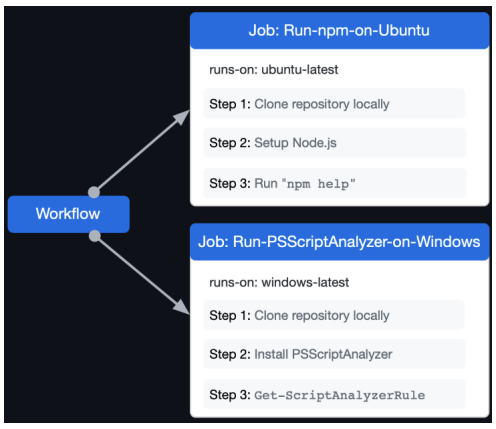

Figura 1.6

1.2.2 Artifacts

Gli artefatti permettono di conservare i dati al termine di un processo e di condividere tali dati con un altro processo nello stesso workflow. Dunque, un artefatto è un file o un insieme di file prodotti durante l'esecuzione di un workflow. Ad esempio, possiamo utilizzare gli artefatti per salvare la *build* e testare *l'output* al termine dell'esecuzione di un workflow.

GitHub archivia i *log* di build e gli artefatti per 90 giorni e questo periodo di conservazione può essere personalizzato. L'archiviazione di artefatti utilizza lo spazio di archiviazione su GitHub. Essi vengono caricati durante l'esecuzione di un workflow ed è possibile visualizzare il nome e le dimensioni di un artefatto nell'interfaccia utente. Quando un artefatto viene scaricato utilizzando l'interfaccia utente di GitHub, tutti i file che sono stati caricati singolarmente come parte dell'elemento vengono compressi insieme in un unico file ***zip***.

GitHub fornisce due azioni che si possono usare per caricare e scaricare gli artefatti della build,

- **Caricamento dei file (*Upload*):** Assegna un nome al file caricato e carica i dati prima che il lavoro finisca.
- **Download dei file:** Possiamo scaricare solo gli artefatti che sono stati caricati durante la stessa esecuzione del workflow. Quando si scarica un file, è possibile fare riferimento ad esso per nome.

È possibile creare un workflow di integrazione continua **(CI)** per creare e testare il codice. L'output della creazione e del test del codice produce spesso file che possiamo utilizzare per eseguire il debug degli errori di test e codice di produzione che possiamo poi distribuire. Si può utilizzare l'azione ***upload-artifact*** per caricare gli artefatti. Quando si carica un artefatto, si può specificare un singolo file o directory o più file o directory. È possibile anche escludere determinati file o *directory* e utilizzare modelli di caratteri *jolly*. Si consiglia di fornire un nome per un artefatto, ma se non viene fornito alcun nome **artifact** verrà utilizzato come nome predefinito.

1.2.3 Environment variables

È possibile utilizzare queste variabili per archiviare le informazioni a cui si desidera fare riferimento nel workflow. Si fa riferimento a *environment variables* all'interno di un passaggio del workflow o di un'azione e le variabili vengono interpolate sulla macchina corridore che esegue il workflow. I comandi eseguiti nelle azioni o nei passaggi del

workflow possono creare, leggere e modificare tali variabili. È possibile impostare le environment variables personalizzate, usare le variabili predefinite che GitHub imposta automaticamente. Queste variabili fanno distinzione tra maiuscole e minuscole.

Per impostare una variabile personalizzata, è necessario definirla nel file del workflow. L'ambito di una variabile personalizzata è limitato all'elemento in cui è definita. È possibile definire environment variables con ambito come segue:

- L'intero workflow, utilizzando **env** al livello superiore del file del workflow. Le variabili sono visibili ed utilizzabili da tutti i job e da tutti gli step.
- Il contenuto di un job all'interno di un workflow, utilizzando "**jobs.<job-id>.env**". Le variabili sono visibili ed utilizzabili da tutti gli step del job in cui sono definite.
- Un passaggio specifico all'interno di un job, utilizzando "**jobs.<job-id>.steps[*].env**". Le variabili sono visibili ed utilizzabili solo dal singolo step in cui sono definite

```yaml
name: Greeting on variable day

on:
  workflow_dispatch

env:
  DAY_OF_WEEK: Monday

jobs:
  greeting_job:
    runs-on: ubuntu-latest
    env:
      Greeting: Hello
    steps:
      - name: "Say Hello Mona it's Monday"
        run: echo "$Greeting $First_Name. Today is $DAY_OF_WEEK!"
        env:
          First_Name: Mona
```

Figura 1.7: Un esempio di definizione di environment variables

1.2.4 Contexts

Oltre alle environment variables, GitHub Actions consente anche di impostare e leggere valori utilizzando i *contexts*. Le variabili di ambiente e i contexts sono destinati all'uso in diversi punti del workflow. Le variabili di ambiente sono sempre interpolate sul runner della macchina virtuale. Tuttavia, parti di un workflow vengono elaborate da GitHub Actions e non vengono inviate al Runner. Non è possibile utilizzare variabili in queste parti di un file del workflow. Invece, possiamo usare i contexts. Ad esempio, un

if condizionale, che determina se un job o un passaggio viene inviato al Runner, viene sempre elaborato da GitHub Actions. È possibile utilizzare un context in un'istruzione *if* condizionale per accedere al valore di una environment variable.

1.3 YAML

YAML è un linguaggio per la serializzazione di dati che viene spesso impiegato per la scrittura dei file di configurazione. All'inizio YAML era l'acronimo di "*Yet Another Markup Language*" (Ancora un altro linguaggio di *markup*), ma poi è stato reinterpretato come "*YAML Ain't Markup Language*" (YAML non è un linguaggio di markup), per sottolinearne il carattere orientato ai dati. Viene considerato un'alternativa a *XML* e *JSON*.

YAML è un linguaggio di programmazione ampiamente diffuso poiché leggibile in chiaro e di facile comprensione. Per comprendere come viene utilizzata la sintassi YAML nella creazione di un file di workflow, in seguito si illustra alcune righe del codice YAML:

```yaml
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

Figura 1.8: Esempio YAML

- **Name :** Specifica il nome del workflow.
- **On :** Specifica il *trigger* per questo workflow. Viene attivata un'esecuzione del workflow ogni volta che un utente esegue il push di una modifica nel Repository o unisce una richiesta pull.
- **Job :** Raggruppa tutti i processi eseguiti nel workflow.
- **Runs-On :** Come anticipato prima, Configura il processo per l'esecuzione sulla versione più recente di un corridore Ubuntu Linux. Ovviamente è possibile scegliere altri corridori Windows e MacOS come specificato in precedenza.

- **Steps :** Raggruppa tutti i passaggi eseguiti nel processo. Ogni elemento contenuto in questa sezione è un'azione separata o uno script di shell.
- **Uses :** Specifica un'azione *built-in* o di terze parti da utilizzare nel workflow, può essere accompagnata da una *keyword* ***with*** presente allo stesso livello la quale specifica i parametri di input dell'azione.
- **Run :** La parola chiave indica al lavoro di eseguire un comando sul corridore.

Uno degli impieghi più comuni di YAML è la scrittura dei file di configurazione. È consigliabile preferire YAML a JSON per la creazione dei file di configurazione perché, anche se nella maggior parte dei casi i due formati sono interscambiabili, YAML è più leggibile e intuitivo.
GitHub Actions utilizza la sintassi YAML per definire il workflow. Ogni workflow viene archiviato come file YAML separato nel repository di codice, in una directory denominata **github/workflows**. La struttura di un documento ".YAML" è essenzialmente la stessa di una mappa (o dizionario), in cui possono essere specificate diverse chiavi a cui assegnare un valore:

```
key: value
another_key: Another value goes here.
```

Figura 1.9

I valori specificati possono essere interi, booleani, stringhe (raggruppate in apici o meno) o anche in notazione esponenziale. Nel caso in cui si voglia assegnare ad una chiave un insieme di valori, è possibile utilizzare i caratteri "|" oppure ">" come nella figura seguente:

```
literal_block: |
    This entire block of text will be the value of the 'literal_block' key,
    with line breaks being preserved.

    The literal continues until de-dented, and the leading indentation is
    stripped.

        Any lines that are 'more-indented' keep the rest of their indentation -
        these lines will be indented by 4 spaces.
folded_style: >
    This entire block of text will be the value of 'folded_style', but this
    time, all newlines will be replaced with a single space.

    Blank lines, like above, are converted to a newline character.

        'More-indented' lines keep their newlines, too -
        this text will appear over two lines.
```

Figura 1.10

1.4 Maven

Build Automation è l'insieme di attività legate all'automatizzazione di alcuni *task* del ciclo di vita del software, quali,

- Compilazione
- Linking
- Esecuzione di test
- Analisi statica
- Creazione di documentazione
- Testing
- .....

Per poter automatizzare queste attività è spesso necessario scrivere del codice in un linguaggio di *scripting* (oppure utilizzare programmi visuali). Tramite un efficace codice di Build Automation, che è rappresentato dal codice del programma, è possibile generare ed effettuare *deploy* automaticamente diverse versioni del programma e reperire automaticamente le risorse necessarie a completare l'esecuzione del programma.

Generalmente per sviluppare un software sono necessarie numerose fasi. Con la build automation l'intero processo viene automatizzato, riducendo il carico di lavoro del programmatore e diminuendo le possibilità di errore da parte dello stesso.

Maven è stato originariamente progettato per semplificare i processi di costruzione nel progetto *Jakarta Turbine*. C'erano diversi progetti e ogni progetto conteneva file di *build ANT* leggermente diversi. I *JAR* sono stati controllati in CVS (*Concurrent Versioning System*). Maven, sviluppato dalla *Apache Software Foundation*, è uno strumento di build automation utilizzato prevalentemente nella gestione di progetti *Java*. Con questo strumento infatti non è più necessaria la compilazione totale del codice, ma viene fatto uso di una struttura di progetto standardizzata su *template* definita **archetype**. Poiché la maggior parte delle configurazioni del progetto sono semplici e riutilizzabili, Maven semplifica la vita degli sviluppatori durante la creazione di *report*, controlli, build e test delle configurazioni di automazione, oltre il *deployment* sui sistemi.

Maven fornisce un comportamento predefinito ragionevole per i progetti. Quando viene creato un progetto Maven, esso crea la struttura del progetto predefinita. Lo sviluppatore deve solo posizionare i file di conseguenza e non ha bisogno di definire alcuna

configurazione in "*pom.xml*". Quest'ultimo, detto anche *Project Object Model* rappresenta la struttura e il contenuto del progetto Maven ed è l'unità fondamentale dell'intero sistema Maven (Nel capitolo successivo vedremo più nel dettaglio). La figura seguente presenta un esempio iniziale di tale file,

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.Alex.Gestione</groupId>
  <artifactId>Gestione</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Gestione</name>
```

Figura 1.11: La struttura iniziale del file pom.xml

Ora consideriamo alcune caratteristiche di Maven:

- Rappresenta un modello completo per i progetti, riutilizzabile, manutenibile e di facile comprensione.
- Sono disponibili i *plug-in* o strumenti che interagiscono con questo modello dichiarativo. Per creare il progetto, Maven fornisce agli sviluppatori opzioni per menzionare gli obiettivi del ciclo di vita e le dipendenze del progetto (che si basano sulle funzionalità del plug-in Maven e sulle sue convenzioni predefinite). Gran parte della gestione del progetto e delle attività relative alla creazione sono gestite dai plug-in Maven. Lo sviluppatore non è tenuto a sapere necessariamente come funzionano i plug-in.
- **Build basate su modelli :** Maven è in grado di creare un numero qualsiasi di progetti in tipi di output predefiniti come *jar, war, metadata*.
- **Gestione del rilascio e pubblicazione della distribuzione :** Senza una configurazione aggiuntiva, Maven si integrerà con il sistema di controllo del codice sorgente come CVS e gestirà il rilascio di un progetto.
- **Build parallele :** Analizza il grafico delle dipendenze del progetto e consente di creare moduli di pianificazione in parallelo. Usando questo, è possibile ottenere miglioramenti delle prestazioni.
- **Migliore segnalazione degli errori e dell'integrità :** Maven ha migliorato la segnalazione degli errori e fornisce un collegamento alla pagina *wiki di Maven* in cui si può ottenere una descrizione completa dell'errore.

Maven è in realtà un *framework* di esecuzione di plug-in in cui ogni attività viene effettivamente eseguita dai plug-in. Essi sono generalmente usati per, creare un file jar, compilare un file di codice, test-unit del codice, creare documentazione di progetto, creare report di progetto ecc.

Capitolo 2

Implementazione dei progetti

L'automazione di un progetto è sempre reso possibile grazie all'ampia disponibilità di strumenti e piattaforme esistenti. Viene di seguito mostrato una serie di passaggi tramite la quale è possibile automatizzare un progetto. Vedremo nel dettaglio i passi svolti in modo da permettere ad un altro utente di poterli replicare con un simile progetto Java. Come prima cosa, analizzeremo in questo capitolo e nel prossimo, uno scenario in cui si crea un nuovo Repository e un progetto Maven inserendo il progetto Java, si carica il codice (il progetto Maven) sul Repository e poi si scrive il file "*.yml*" per ottenere gli stessi risultati seguenti. Vedremo anche come sarà possibile effettuare il *fork* di un progetto già esistente per poi modificare cambiando il codice e facendolo funzionare su di un altro progetto.

2.1 Il progetto Java

Come esempio, usiamo un progetto, sviluppato con *Eclipse*, per la gestione di una società che organizza pacchetti turistici in città europee. Con la piattaforma interagiscono un amministratore della piattaforma, i clienti (turisti), e gli operatori turistici (gestori di hotel e di servizi di ristorazione, trasportatori, guide turistiche). L'amministratore, autenticato in base a *username* e *password*, può creare nuovi pacchetti di visite turistiche e visualizzare le sottoscrizioni da parte dei turisti. I nuovi pacchetti sono creati aggiungendo un codice univoco, una descrizione, la città di riferimento, il numero massimo di partecipanti. Gli operatori turistici si pre-registrano fornendo le loro generalità (nome della società, partita iva, città), username e password. Ciascun operatore ha un ruolo che può essere, per esempio, Operatore Hotel, Operatore ristorazione, Trasportatore, Guida turistica ecc.. Una volta registrati, gli operatori possono collegarsi alla piattaforma per aggiungere servizi ad un pacchetto. Gli Operatori Hotel possono inserire in un pacchetto un "Servizio Hotel", gli operatori viaggio possono inserire un "Servizio Trasporto", gli operatori ristorazione possono inserire un "Pasto", le guide turistiche possono in-

serire una “Visita museale”. I turisti possono visualizzare la lista dei pacchetti turistici (dopo l’autenticazione), eventualmente effettuando ricerche in base alla città e possono acquistare un pacchetto (se registrati ed autenticati). La struttura del progetto è seguente,

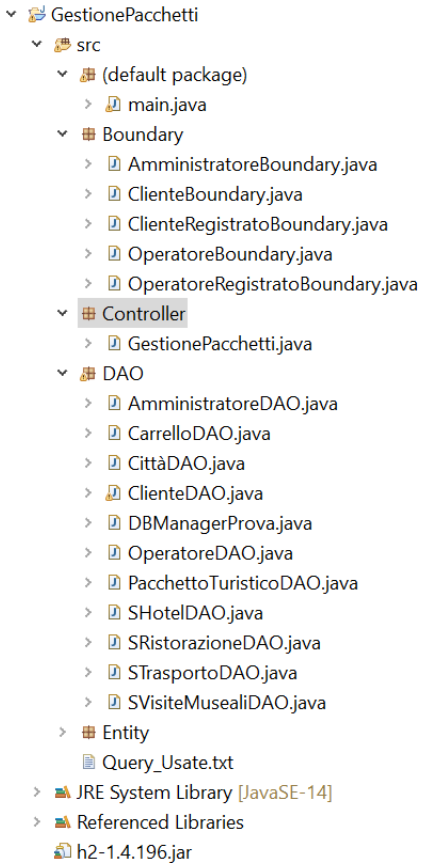

Figura 2.1: Il progetto Java utilizzato

2.2 Il progetto Maven

Il primo passo da fare è la scelta di uno strumento del cosiddetto *build automation* che permetta di gestire in modo efficace le dipendenze e l’utilizzo di plug-in esterni all’interno di un progetto. Per utilizzare un tale strumento è necessario conoscerne almeno le caratteristiche di base ed il funzionamento. Nel nostro caso, usiamo *Maven*, che è uno dei strumenti principali di *build automation*. Eclipse integra nativamente Maven, per cui potremo gestire tutte le fasi del progetto direttamente da Eclipse.

Adesso andiamo a creare un progetto Maven su Eclipse facendo ***File -> New -> other -> Maven -> Maven Project***. Apparirà una finestra come nella figura seguente,

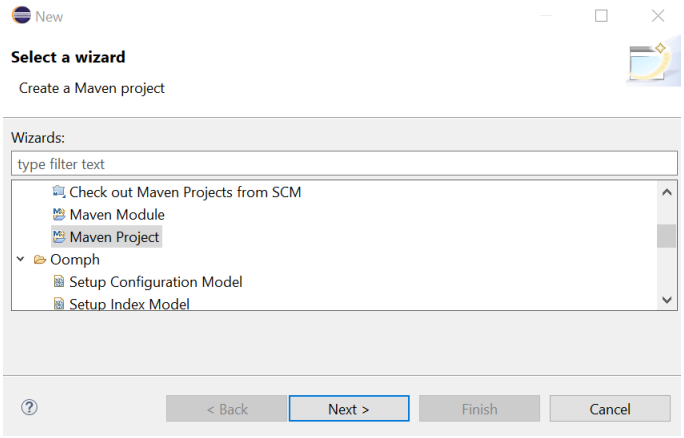

Figura 2.2: New Maven Project

Dopo aver premuto ***Next***, ci viene data la possibilità di selezionare diverse opzioni del nostro workspace, ma possiamo tranquillamente passare alla prossima schermata (figura 2.3) dopo aver selezionato la casella ***Create a simple Project***.

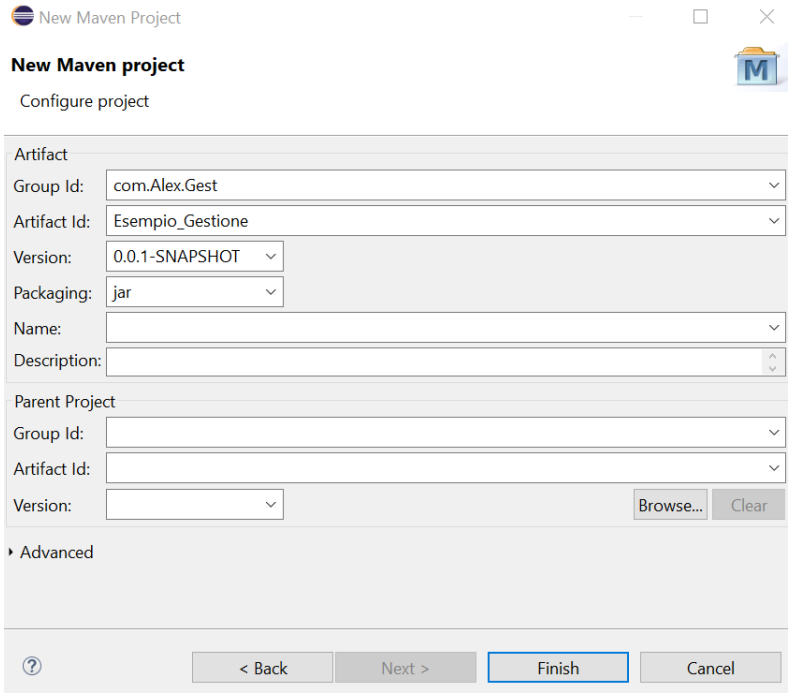

Figura 2.3

Group-id rappresenta un ID del gruppo del progetto, questo è generalmente unico all'interno di un'organizzazione o di un progetto e *Artifact-id* rappresenta un ID del

progetto, questo è generalmente il nome del progetto. Possiamo compilare questi due campi, per esempio, come fatto nel *Window* precedente. Una volta creati il group-id e artifact-id, verranno creati tutti i pacchetti combinando questi due campi. Inoltre ci viene data la possibilità di scegliere il formato in cui verranno creati i pacchetti, nel mio caso scelgo **jar** (è anche l'opzione di *default*) e la versione del *Artifact* è **0.1-SNAPSHOT** per *default*. I campi che si trovano subito dopo sono facoltativi: se vogliamo ereditare alcune dipendenze da un altro progetto, possiamo specificarlo nella sezione **Parent Project**, ma in questo caso non ci sarà bisogno. Premendo ***Finish*** viene creato il progetto Maven.

A questo punto possiamo visualizzare nella sezione di progetti di Eclipse, il nuovo progetto Maven che è stato appena creato,

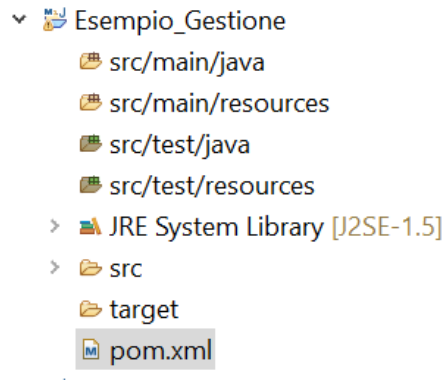

Figura 2.4: La struttura iniziale del progetto Maven

Andiamo ad analizzare la struttura e le cartelle del progetto,

- **src/main/java -** Questo file contiene principalmente il codice sorgente del progetto Java, per cui è riservato allo sviluppatore del codice.
- **src/main/resources -** Questo file contiene le eventuali risorse utilizzate dal codice sorgente (*third party files*).
- **src/test/java -** Una volta che lo sviluppatore completa il progetto, tale codice va testato mediante dei opportuni *test cases*. Questo file contiene appunto i test cases creati per esempio con *JUnit, testng* ecc per verificare la correttezza dei risultati.
- **src/test/resources -** Ha la stessa funzione di *main/resources*. Vengono mantenute le risorse utili per test cases.
- **src e target -** Al tempo di esecuzione questi due file saranno utilizzati da Maven per contenere dei file che vengono creati temporaneamente.

- **pom.xml -** Questo file ha il compito più importante di tutti. Vediamo in dettaglio di che si tratta.

2.2.1 POM.XML

POM sta per *Project Object Model*, è l'unità di lavoro fondamentale in Maven. È un file *XML* che risiede nella directory di base del progetto come "*pom.xml*". Quest'ultimo contiene le informazioni sul progetto e vari dettagli di configurazione utilizzati da Maven per costruire i progetti. Contiene anche gli obiettivi e i *plug-in*. Durante l'esecuzione di un *task* o di un *goal*, Maven cerca il POM nella directory corrente. Legge il POM, ottiene le informazioni di configurazione necessarie e quindi esegue il goal. Alcune delle configurazioni che possono essere specificate nel POM sono le seguenti,

- *project dependencies*
- *plugins*
- *goals*
- *build profiles*
- *project version*
- *developers*

Nel nostro caso il file "*pom.xml*" iniziale è il seguente,

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.Alex.Gest</groupId>
  <artifactId>Esempio_Gestione</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

Figura 2.5: pom.xml

Tutti i file POM richiedono l'elemento **project** e i tre campi obbligatori: ***groupId, artifactId, version*** . Il *tag* progetto specifica le impostazioni dello schema di base come lo schema di *Apache* e la specifica "*w3.org*". Più avanti vedremo come viene modificato ulteriormente il file "*pom.xml*" in base alle nostre esigenze.

2.2.2 Test Case

Il test, come visto precedentemente, è il processo di controllo di un'applicazione che funziona come previsto: per "funzionare come previsto" s'intende che, per un input noto, deve dare l'output previsto. In altre parole, il test è un processo di verifica e convalida. Per definire i test cases usiamo **JUnit** che è un *framework* di unit test *open source* per programmatori java. Viene utilizzato solo per unit test. Il test di integrazione viene eseguito da *TestNG*.
Dopo aver copiato il nostro progetto Java all'interno della cartella *src/main/java* e le eventuali risorse utilizzate nella cartella *src/main/resources* (**Attenzione :** *Dopo qualsiasi aggiornamento del progetto Maven, si ricorda di fare update cliccando tasto destro su pom.xml -> Maven -> update project*), andiamo a creare i test cases che ci servono. Quindi facciamo, tasto destro su **src/test/java** -> **New** -> **JUnit Case**. Si aprirà la seguente finestra,

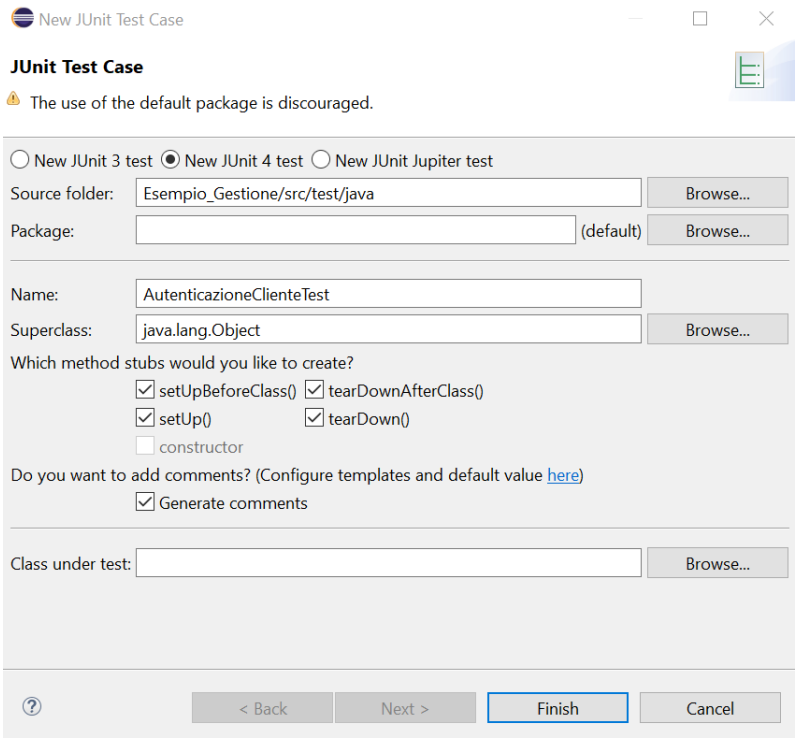

Figura 2.6: New JUnit Test Case

Il nome del test case sarà, per esempio, "*AutenticazioneClienteTest*", poi possiamo cliccare su ***Finish***. Dopo aver aggiunto i test che mi servono, il *JUnit Test Case* avrà la seguente forma (è da notare che JUnit fornisce la classe *Assert* per verificare determinate condizioni. I metodi della classe *Assert* confrontano il valore di output con il valore previsto),

```java
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import Entity.ClienteRegistrato;

/**

/**
 * @author laksh
 *
 */
public class AutenticazioneClienteTest {

    private ClienteRegistrato clienteTest;

    @Before
    public void setUp() throws Exception {
        clienteTest = new ClienteRegistrato("Mario", "Rossi", "admin", "admin", "MarioRossi@gmail.com");
    }

    /**
     * @throws java.lang.Exception
     */

    @Test
    public void getNomeTest() {
         assertEquals("Mario",clienteTest.getNome());
    }

    @Test
    public void getCognomeTest() {
         assertEquals("Rossi",clienteTest.getCognome());
    }

    @Test
    public void getUsernameTest() {
         assertEquals("admin",clienteTest.getUsername());
    }

    @Test
    public void getPasswordTest() {
         assertEquals("admin",clienteTest.getPassword());
    }

    @Test
    public void getEmailTest() {
         assertEquals("MarioRossi@gmail.com",clienteTest.getEmail());
    }


}
```

Figura 2.7: Esempio di un Test Case per verificare i dati di autenticazione di un dato client

2.2.3 Ulteriori modifiche del pom.xml

A questo punto dobbiamo definire delle dipendenze (**<dependences>**) all'interno del file "*pom.xml*" che permettono di acquisire le librerie necessarie per eseguire i test cases e verranno conservate nella cartella **Maven Dependencies** che si crea automaticamente ogni volta che il file POM viene salvato (Tali librerie avranno un loro *tag dependency* che le identifica e che permette a Maven di scaricarle dal Repository centrale). Nel nostro caso, tale file, dopo aver specificato quali sono le dipendenze di cui abbaimo bisogno, avrà la seguente struttura,

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.Alex.Gest</groupId>
  <artifactId>Esempio_Gestione</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

```
    <!-- https://mvnrepository.com/artifact/org.testng/testng -->
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.3.0</version>
    <scope>test</scope>
</dependency>
```

```
    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.30</version>
</dependency>
```

```
          <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.6</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>com.jgoodies</groupId>
            <artifactId>jgoodies-common</artifactId>
            <version>1.8.0</version>
        </dependency>
```

```
        <dependency>
            <groupId>com.jgoodies</groupId>
            <artifactId>jgoodies-looks</artifactId>
            <version>2.7.0</version>
        </dependency>

        <dependency>
            <groupId>com.jgoodies</groupId>
            <artifactId>jgoodies-forms</artifactId>
            <version>1.8.0</version>
        </dependency>



    </dependencies>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>



    <build>
    <pluginManagement>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M7</version>
        </plugin>
    </plugins>
    </pluginManagement>
    </build>

</project>
```

Figura 2.8: pom.xml finale

Dove:

- *TestNG* è un framework di test ispirato a JUnit ma che introduce alcune nuove funzionalità che lo rendono più potente e facile da usare. Supporta test configurati da annotazioni, test basati sui dati, test parametrici, ecc.
- *MySQL Connector/J* è un driver *JDBC* di tipo 4, il che significa che è una pura implementazione Java del protocollo *MySQL* e non si basa sulle librerie *client*

MySQL. Questo driver supporta la registrazione automatica con *Driver Manager*, controlli di validità standardizzati, *SQLExceptions* categorizzate, supporto per grandi conteggi di aggiornamenti, supporto per varianti data-ora locali e *offset* dal pacchetto *java.time* ecc

- *JUnit* è un framework di unit test per scrivere ed eseguire test automatici ripetibili su Java.
- La libreria *JGoodies Common* fornisce codice di convenienza per altre librerie e applicazioni *JGoodies*.
- Il *maven-compiler-plugin* viene utilizzato per compilare i sorgenti del progetto.

Abbiamo finito di creare il progetto Maven ed è pronto per essere caricato sul Repository remoto su GitHub. Per eseguire il progetto creato in locale possiamo cliccare il tasto destro su "*pom.xml*" e poi **Run**. Verrà visualizzata una lista di *goals* che possono essere eseguiti, tra cui quelli principali: *build, clean, install, test*. Cliccando su build, è possibile specificare esplicitamete il goal che vogliamo eseguire invece di sceglierli dalla lista. Per esempio facciamo ***pom.xml -> Run -> build*** e scriviamo ***compile***. Il risultato è seguente,

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< com.Alex.Gest:Esempio_Gestione >-------------------
[INFO] Building Esempio_Gestione 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ Esempio_Gestione ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ Esempio_Gestione ---
[INFO] Nothing to compile - all classes are up to date
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.983 s
[INFO] Finished at: 2022-08-11T19:17:48+02:00
[INFO] ------------------------------------------------------------------------
```

Figura 2.9: L'output di compile

Allo stesso modo è possibile eseguire **install** (crea degli oggetti nella cartella *target*) e i **test**. La parola chiave **clean** invece permette di eliminare tutti i oggetti creati mediante *install*.

Abbiamo visto la struttura iniziale del progetto Maven nella figura "2.4". La struttura finale è seguente,

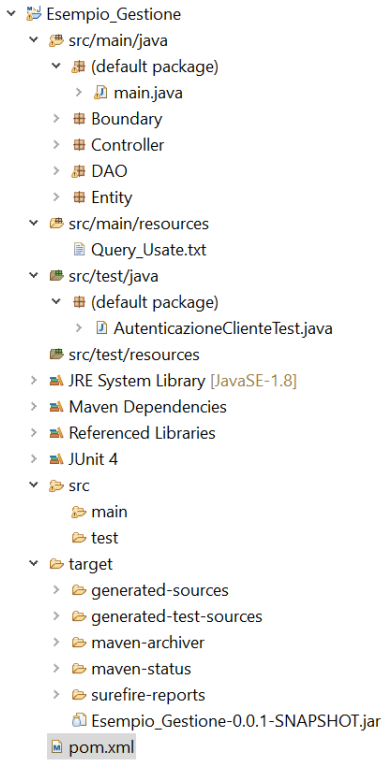

Figura 2.10: La struttura finale del progetto Maven

Capitolo 3

Automazione dei progetti

In questo capitolo, come indicato in precedenza, ci occuperemo del trasferimento dei progetti che sono stati creati nel capitolo precedente in un Repository di GitHub Actions remoto, attraverso il quale sarà possibile condividere, migliorare il codice ma sopratutto automatizzare vari fasi di sviluppo software, tra cui il *Testing*.

3.1 La creazione di un Repository e il caricamento del progetto

Come abbiamo visto precedentemente, il Repository è un contenitore di *storage* di tutta la storia del codice e degli altri artefatti di un progetto, durante tutto il suo ciclo di vita. La questione che qui va affrontata, è come crare un nuovo Repository GitHub per poi caricare un progetto Maven esistente su di esso, per apportare delle modifiche da parte di più contributori, quindi per poter automatizzare il progetto.

Innanzitutto andiamo sul sito ***https://github.com/*** per poi effettuare *sign in* se disponiamo di un *GitHub Account* (altrimenti bisogna creare un nuovo Account con le proprie credenziali). Nel *HomePage* di GitHub ci viene data la possibilità di creare un nuovo Repository tramite ***create a new repository***.

Si specifica il nome del Repository (es: *Example_Gestione*) come illustrato nella figura seguente. Si tratta di un Repository pubblico e si richiede di avere a disposizione il file README (cliccando sulla casella corrispondente) per le eventuali descrizioni sul progetto. Per proseguire, clicchiamo sul ***create Repository*** e ci viene indirizzato al Repository appena creato in cui al momento è presente solamente il file README.

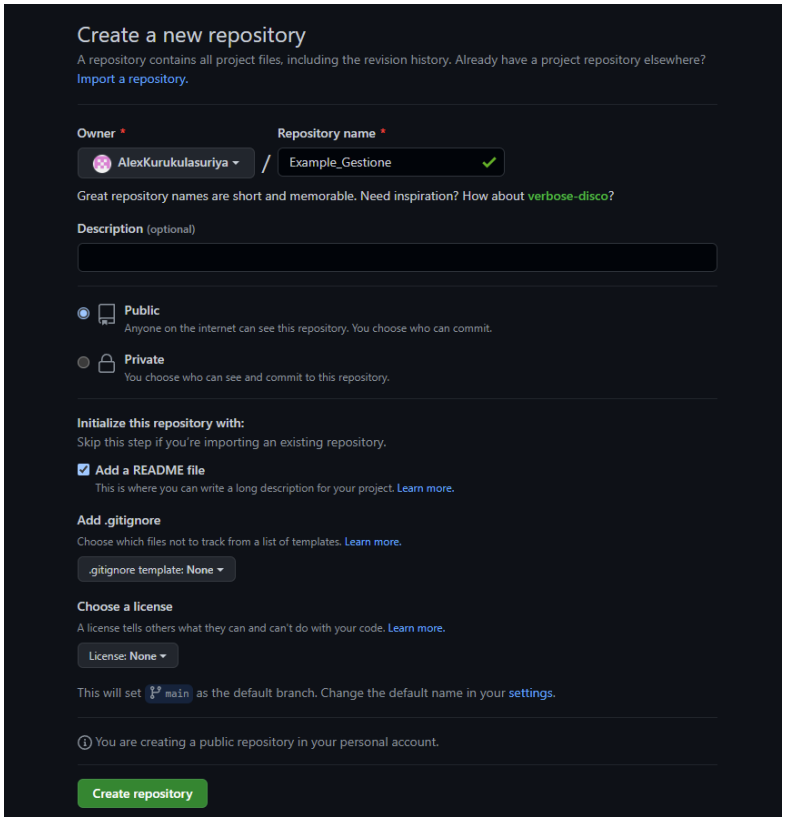

Figura 3.1: New Repository

A questo punto andiamo sul sito ***https://git-scm.com/downloads*** e scarichiamo *git* per il proprio sistema operativo *(Il protocollo git è particolarmente utilizzato in progetti open source distributi sul Web. GitHub è una delle piattaforme che mettono a disposizione funzionalità di hosting tramite git)*. Dopo la fase di installazione di tale programma, è opportuno creare una cartella nel *path*, nel mio caso, "*C:/ Users/ laksh/ git*" (sul disco) e la chiamo, per esempio, "*AutomationGestione*". Aprendo questa cartella, bisogna fare ***tasto destro -> Git Bash Here***.
E ora andiamo sul nostro Repository per copiare l'URL, come nella figura seguente,

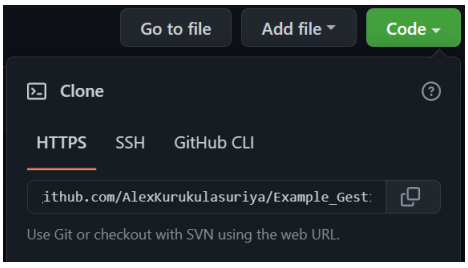

Figura 3.2: URL del mio Repository

Eseguiamo attentamente i seguenti passi sul Git Bash,

- Incollare *l'URL* precedentemente copiato per poter clonare il *Repository GitHub* sul PC locale (digitare ***git clone https://github.com/AlexKurukulasuriya/Example_Gestione.git***),

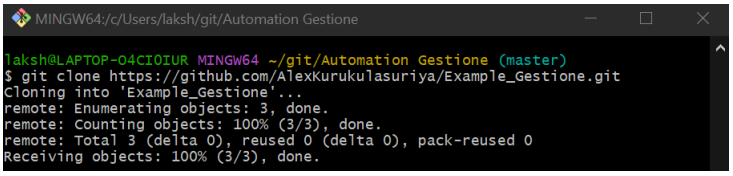

Figura 3.3

A questo punto viene creato il *clone* del Repository *Example_Gestione* all'interno della cartella precedentemente creata: *AutomationGestione*.

- Spostarsi nella cartella *Example_Gestione* (digitare ***cd Example_Gestione/***),

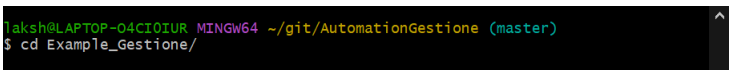

Figura 3.4

- Andare nella cartella in cui è presente il progetto Maven (*Esempio_Gestione*), copiare tutti i file, incollare i file copiati nella cartella *Example_Gestione*, come mostrato nella figura,

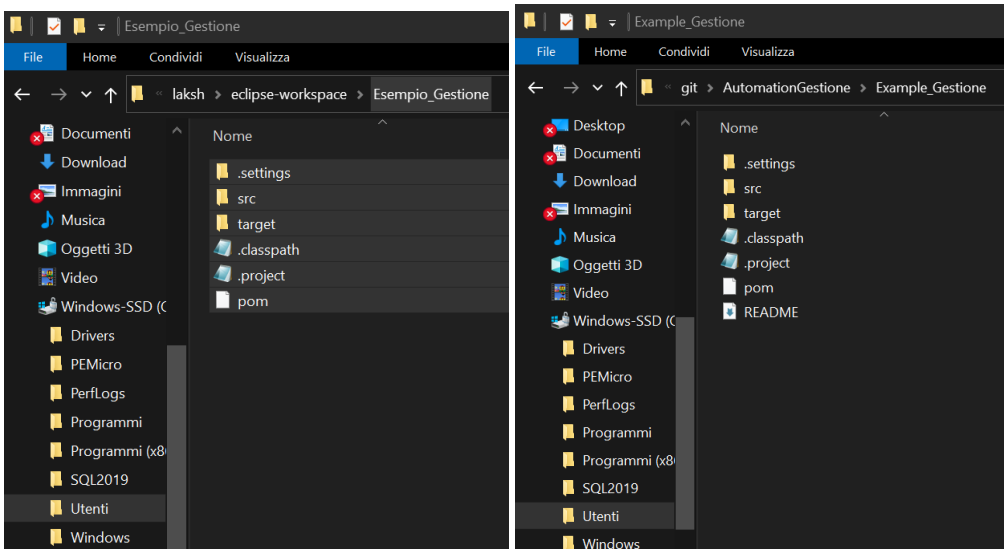

Figura 3.5

- Digitare su Git Bash: "***git add .***" per aggiungere sul Repository tutte le modifiche,

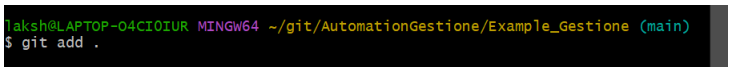

Figura 3.6

- Effettuare il ***commit*** digitando sul Git Bash il comando: ***git commit -m "Adding the project"*** . (La frase contenuta tra doppi apici indica il tipo di commit che si sta eseguendo).
- Per concludere, digitare ***git push*** sul Git Bash,

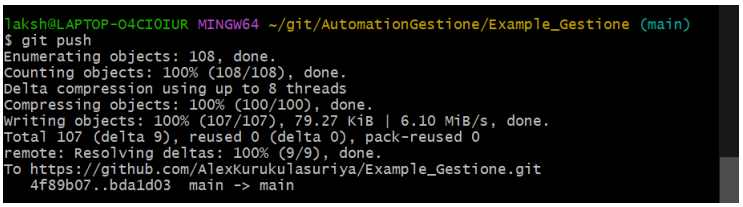

Figura 3.7

Infine nella figura seguente possiamo osservare la struttura del Repository finale (*La sequenza di passi descritti precedentemente è uno dei tanti modi per caricare un progetto Maven sul Repository GitHub. Si può effettuare il trasferimento tramite Eclipse stesso, GitHub Dekstop, Sublime Merge ecc ottenendo il medesimo risultato*),

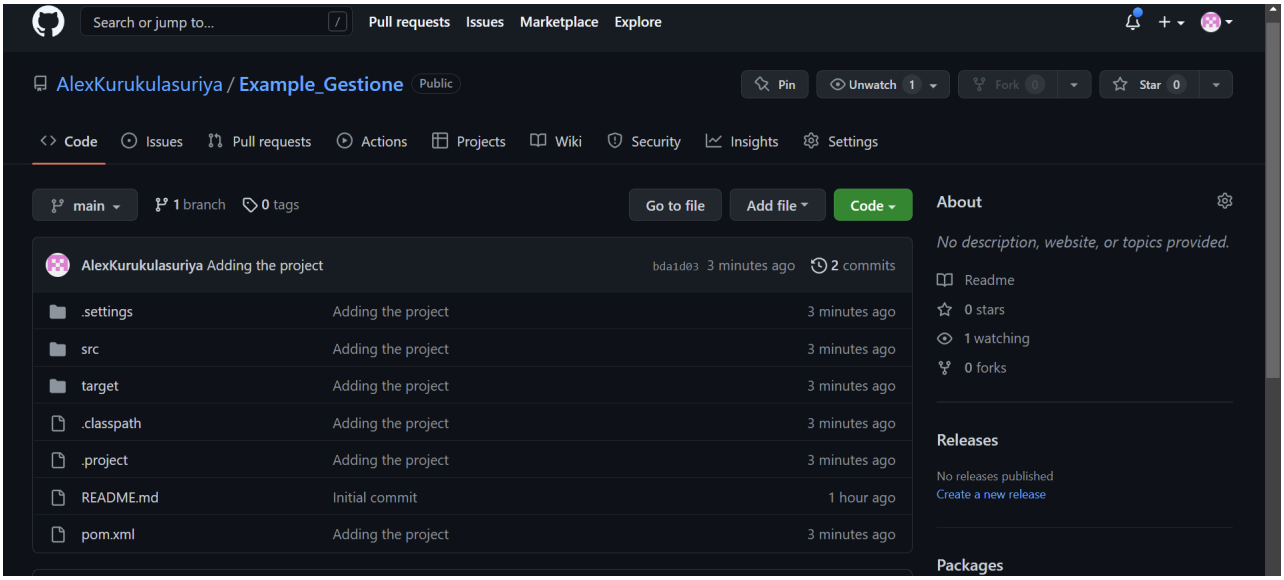

Figura 3.8: Repository che si ottiene dopo aver caricato il progetto Maven su GitHub

3.2 Definire la pipeline

Il primo passo per creare una pipeline CI *(Integrazione Continua)* con GitHub Actions è creare o scegliere un Repository su GitHub. Si può usare una base di codice di progetto esistente, eseguire il fork di un progetto su GitHub o iniziare da zero.

Una pipeline CI viene eseguita quando il codice viene modificato e deve assicurarsi che tutte le modifiche funzionino con il resto del codice quando è integrato. Dovrebbe anche compilare il codice, eseguire test e verificare che sia funzionale. Una pipeline CD *(Distrubuzione Continua)* fa un ulteriore passo in avanti e distribuisce il codice incorporato in produzione.

Per iniziare a creare la pipeline *CI/CD*, aprire la scheda Azioni (*Actions*) GitHub nella barra di navigazione superiore del Repository,

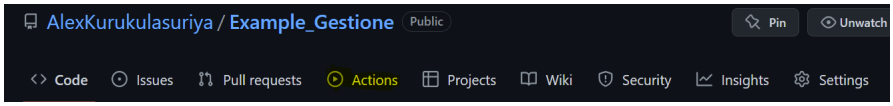

Figura 3.9

A questo punto è possibile vedere un elenco di modelli di automazione CI/CD e del workflow che corrispondono alla tecnologia utilizzata dal proprio progetto (dei *template* di base da cui poter partire per definire il proprio workflow) come si può vedere nella figura seguente,

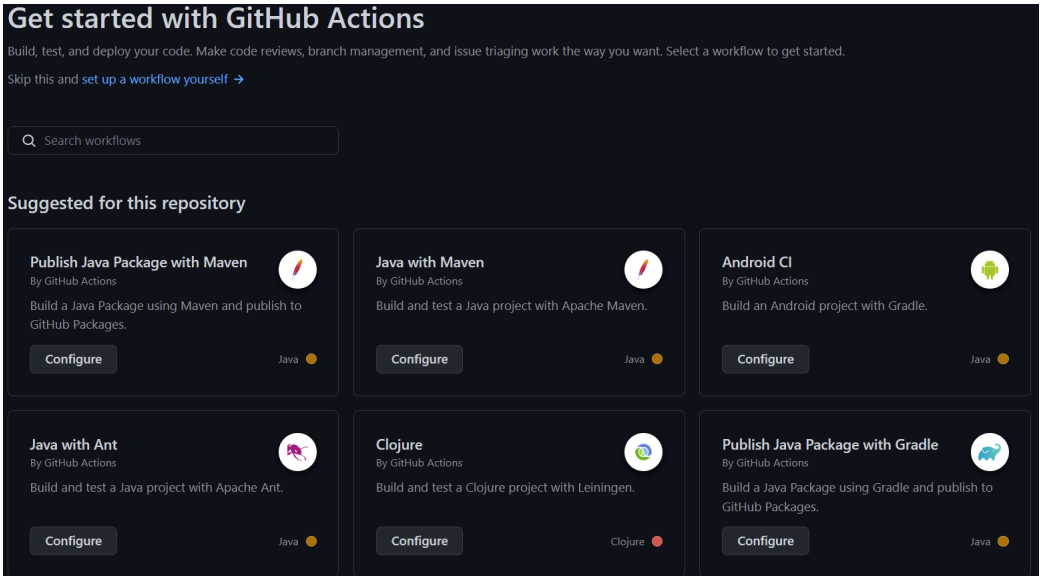

Figura 3.10: Template per i workflow

Non è obbligatorio usare un template ma può risultare utile per avere una base di partenza. Nel nostro caso ci conviene scegliere ***Java With Maven***, quindi procediamo per configurare il file *"maven.yml"* che si crea cliccando su *Java With Maven* per poi affidare ad esso le varie fasi di sviluppo (compilazione, esecuzione dei test, packaging ecc) ai job del workflow. La struttura finale di tale file è seguente,

```yaml
name: Java CI with Maven

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Setup Maven Action
      uses: s4u/setup-maven-action@v1.2.1
      with:
        java-version: '11'
        maven-version: '3.8.3'
    - name: Build with Maven
      run: mvn compile


  Test:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - name: Setup Maven Action
        uses: s4u/setup-maven-action@v1.2.1
        with:
          java-version: '11'
          maven-version: '3.8.3'
      - name: Running Test
        run: mvn test

  Package:
    runs-on: ubuntu-latest
    needs: Test
    steps:
      - name: Setup Maven Action
        uses: s4u/setup-maven-action@v1.2.1
        with:
          java-version: '11'
          maven-version: '3.8.3'
      - name: Package
        run: mvn -D skipTests clean package
      - name: Create Artifact
        if: success()
        uses: actions/upload-artifact@v3
        with:
          name: Esempio_Gestione
          path: target/*.jar
```

```yaml
Release:
  runs-on: ubuntu-latest
  needs: Package
  steps:
    - name: Download file Jar Artifact
      uses: actions/download-artifact@v3
      with:
        name: Esempio_Gestione
    - name: Create Release
      uses: marvinpinto/action-automatic-releases@v1.2.1
      with:
        repo_token: ${{ secrets.GITHUB_TOKEN }}
        prerelease: false
        automatic_release_tag: latest
        files: Esempio_Gestione

Deployment:
  runs-on: ubuntu-latest
  environment: production
  steps:
    - name: deploy
      run: echo "deploy"
```

Figura 3.11: maven.yml

Per quanto riguarda la fase di *Deployment*, la *Continuous deployment (CD)* è la pratica di utilizzo dell'automazione per pubblicare e distribuire (*deploy*) gli aggiornamenti software. Come parte del tipico processo CD, il codice viene automaticamente compilato e testato prima della distribuzione. È possibile configurare un *workflow GitHub Actions* per distribuire il prodotto software. Per verificare che il prodotto funzioni come previsto, il workflow può compilare il codice nel Repository ed eseguire i test prima del deployment (La distribuzione continua è spesso abbinata all'integrazione continua).
Ora possiamo effettuare il *commit* delle modifiche cliccando su ***Start Commit-> Commit Changes***,

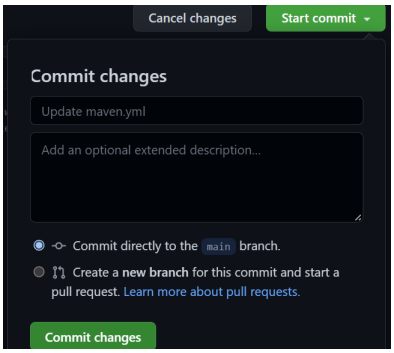

Figura 3.12: commit

Il commit darà luogo alla cartella ***".github/workflows"*** sul Repository principale, all'interno della quale si troverà il file *"maven.yml"*.

Una volta effettuato il *commit*, andiamo nella sezione *Actions* dove sarà possibile visualizzare lo schema seguente (qui è possibile notare le fasi di sviluppo di un software che sono stati specificati nel file "*maven.yml*" e cliccando su ogni fase, si può vedere nel dettaglio gli *step* seguiti passo dopo passo),

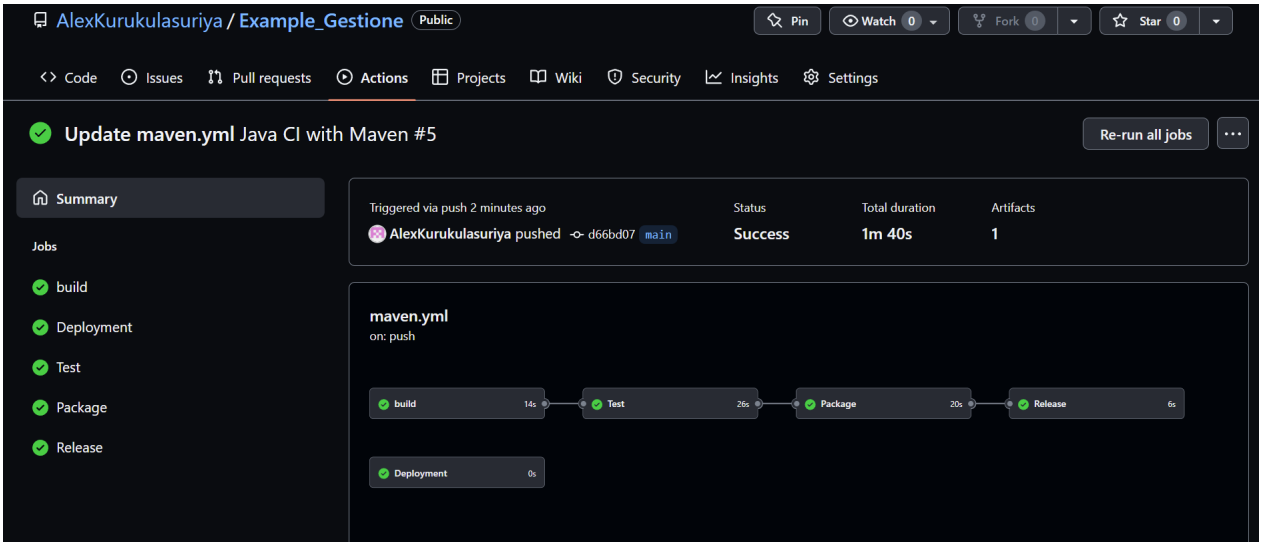

Figura 3.13: Il risultato del commit nella sezione Actions

3.3 Packaging

Una fase di creazione del pacchetto(*Packaging*) è una parte comune di un workflow di integrazione continua(CI). Dopo aver compilato e testato il codice, un passaggio di creazione del pacchetto può produrre un elemento eseguibile o distribuibile. A seconda del tipo di applicazione che si sta creando, questo pacchetto può essere scaricato localmente per il test manuale, reso disponibile per il *download* da parte degli utenti o distribuito in un ambiente di gestione temporanea o di produzione. Ad esempio, un workflow di integrazione continua per un progetto Java può essere eseguito per produrre un file *JAR*.

Anche nel nostro caso sarà possibile esaminare l'esecuzione del workflow (come nella figura 3.13) e scaricare l'elemento prodotto.

Figura 3.14: Artifact prodotto

Ciò consentirà di eseguire il codice nella richiesta pull sul computer locale, che può aiutare con il *debug* o il test della richiesta pull.
Un'altra caratteristica di GitHub è che, esso funge da servizio di *hosting* di pacchetti. Si può scegliere di condividere i prorpri pacchetti con tutto GitHub o pacchetti privati da condividere con collaboratori o un'organizzazione. Pubblicare i pacchetti su GitHub consentirà agli sviluppatori del progetto di essere sempre in grado di eseguire e testare facilmente l'ultima *build* dal *branch* predefinito, installandola dal GitHub.

3.4 Eseguire Fork

Un *fork* è una copia di un Repository già esistente che è possibile gestire in locale, consentono di apportare delle modifiche a un progetto senza influire sul Repository originale. Si può inoltre recuperare gli aggiornamenti o inviare modifiche al Repository originale con richieste pull. Cioè, i fork vengono utilizzati per proporre modifiche al progetto di qualcun altro (a cui non si dispone dell'accesso in scrittura) o per utilizzare il suo progetto come punto di partenza per la propria idea poiché condividendo il codice, possiamo creare software migliori e più affidabili.

Nel caso in cui effettuiamo il fork al progetto di un altro utente per apportare delle modifiche, si apre una richiesta *pull* che confronta il proprio lavoro con il *Repository upstream* (Repository originale). È possibile concedere a chiunque abbia accesso push al Repository upstream, l'autorizzazione per inviare le modifiche al *branch* della richiesta *pull* (compresa l'eliminazione del branch) . Ciò accelera la collaborazione consentendo ai manutentori del Repository la possibilità di eseguire *commit* o eseguire test localmente sul branch della richiesta *pull* da un fork di proprietà dell'utente prima dell'unione. Quindi possiamo seguire il fork del Repository, fare la correzione e inviare una richiesta pull al proprietario del progetto.

Per eseguire il fork, bisogna prima passare a un Repository pubblico che si desidera modificare o usare come modello di partenza. Nel mio caso considero un Repository esempio messo a disposizione da GitHub su cui è possibile effettuare il fork,

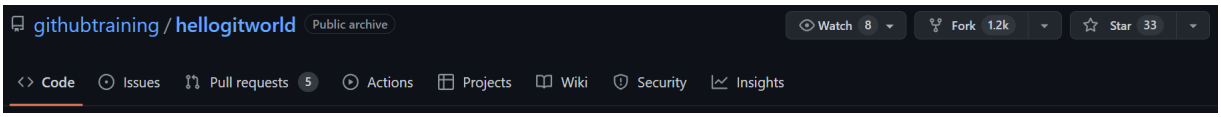

Figura 3.15: githubtraining/hellogitworld

- Il metodo più semplice per eseguire il fork di un Repository è fare *clic* sul pulsante "***Fork***" in alto a destra, che creerà automaticamente un nuovo Repository nel proprio *account*,

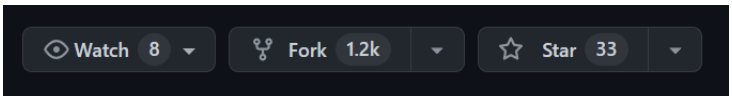

Figura 3.16: Pulsante "*Fork*"

- Selezionare un proprietario per il Repository fork e per impostazione predefinita, i fork hanno lo stesso nome dei Repository principali (è possibile cambiare il nome del fork per distinguerlo ulteriormente),

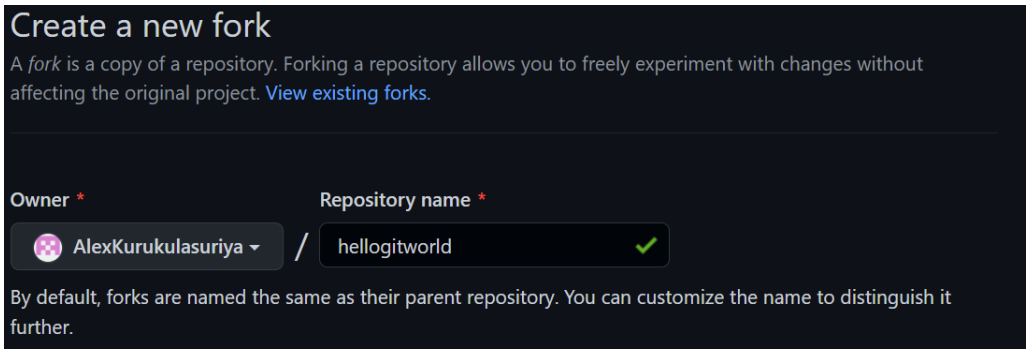

Figura 3.17

- Si può aggiungere una descrizione del fork nella sezione *Description* (Facoltativamente) e poi selezionare se copiare solo il *branch* predefinito o tutti i *branch* nel nuovo fork. Per molti scenari di fork, come il contributo a progetti *open source*, bisogna solo copiare il *branch* predefinito (impostazione di default),

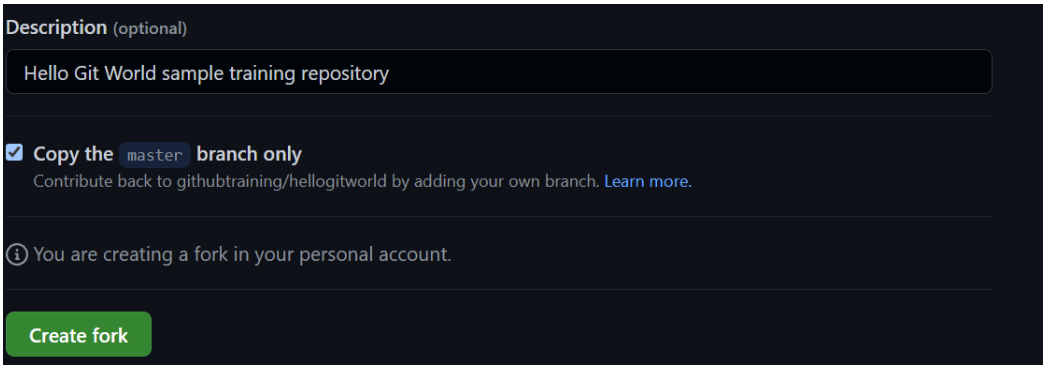

Figura 3.18

- Infine fare click su ***Create Fork***.

A questo punto abbiamo a disposizione una coppia del Repository da modificare ma non abbiamo i file di quel Repository localmente sul computer cosi da poter apportare le nostre modifiche sul progetto.

3.4.1 Clonare il nuovo Repository

Abbiamo visto in precedenza come si carica un progetto Maven su un Repository tramite *Git Bash*. Anche qui i passi da seguire sono simili,

- Andiamo sul Repository nuovo per copiare l'URL, come nella figura seguente,

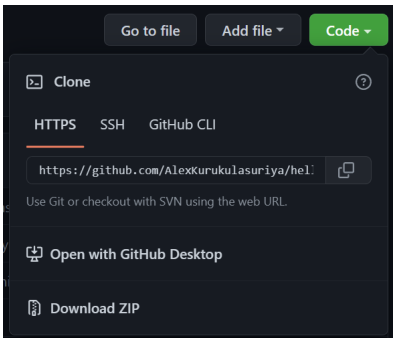

Figura 3.19: URL del nuovo repository fork

- Creare una cartella sul disco locale e aprire il Git Bash (come abbiamo visto prima per *l'update project*).
- Digitare su Git Bash: ***git clone https://github.com/AlexKurukulasuriya/hellogitworld.git***,

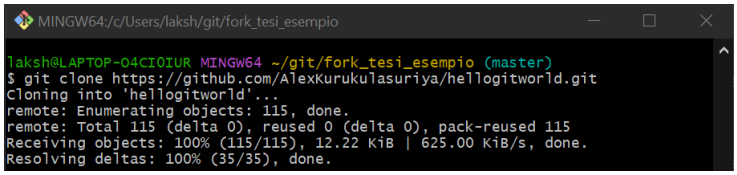

Figura 3.20

Ora sarà possibile visualizzare nella cartella creata in precedenza un nuovo file corrispondente al nostro *Repository fork*.

Chiaramente ognuno è libero di scegliere un Repository specifico su cui lavorare. Dopo aver scelto tale Repository, i passi da seguire per il fork e per clonare i file

sono quelli che abbiamo visto poco fa. A questo punto avendo abbastanza familiarità con GitHub e altri strumenti, sarà possibile modificare il *Repository fork* apportando delle modifiche personali (inserendo un progetto diverso magari). *Come menzionato in precedenza, la sequenza di passi descritti qui sopra è uno dei tanti modi per clonare un Repository GitHub. Si può utilizzare altri strumenti come GitHub Dekstop, Sublime Merge ecc attraverso cui è possibile arrivare allo stesso risultato.*

Conclusione

Ogni software passa attraverso un ciclo di vita di sviluppo per soddisfare i requisiti dei clienti di un prodotto di alta qualità noto come **SDLC** (*Software Development Life Cycle*). Nell'industria del software in crescita, gli sviluppatori competono per produrre software di alta qualità pur rimanendo entro la loro gamma di costi e limiti di tempo. Automatizzare tali processi aiuta a raggiungere gli obiettivi che abbiamo menzionato in questo elaborato con il minimo di manodopera, tempo e costi, pur mantenendo un alto livello di produttività ed efficienza.

La fase più consigliata per l'automazione è l'ambiente di test. I test consentono ai team di verificare la funzionalità dell'applicazione segnalando, monitorando e correggendo i *bug* fino a quando non soddisfa gli *standard* qualificati. Uno dei suoi vantaggi è che gli strumenti di test offrono una funzione di riutilizzabilità come abbiamo osservato in precedenza. Permette di risparmiare tempo consentendo l'implementazione immediata in varie aree dell'applicazione. Inoltre, la diminuzione dell'input umano manuale aumenta la precisione e l'efficienza. Pertanto, l'automazione dei test fornisce un *feedback* efficiente, crea nuove funzionalità e migliora la produttività dello sviluppatore. Integrazione continua e distribuzione continua *(CI/CD)* consentono di automatizzare la *pipeline* di compilazione, test e distribuzione. Come è stato indicato nei capitoli precedenti, sono disponibili vari strumenti che possono essere utilizzati per il processo di automazione di *Testing*, per il *hosting* dei progetti e del codice, i *tool* di *build automation*.

Build automation permette di gestire in modo efficace le dipendenze e l'utilizzo di *plug-in* esterni all'interno di un progetto. Per utilizzare un strumento di *build automation* è necessario conoscerne almeno le caratteristiche di base ed il funzionamento. Nel nostro caso, usiamo *Maven*, facendo uso anche di *YAML* che viene utilizzato in molte dizioni/dialetti differenti a seconda della piattaforma utilizzata (Esistono altre piattaforme che usano linguaggi differenti per specificare la *pipeline*).

Poichè i software vengono sviluppati in *team*, è fondamentale una comunicazione solida ed affidabile: Gli strumenti che abbiamo analizzato in questi capitoli consento-

no un buon grado di visibilità e descrizione del progetto nella fase di sviluppo. Ogni contributore è libero di effettuare il *fork* di un progetto esistente per poi apportare dei miglioramenti oppure di adattare tale progetto per uno scopo personale. Si può suggerire delle modifiche al progetto mediante le richieste *pull* che verranno accettate dallo sviluppatore nel caso in cui rappresentino un miglioramento. Ma sopratutto si può dar luogo ad una nuova idea utilizzando gli strumenti a disposizione, come illustrato nei capitoli precedenti, che ci permettono di accelerare l'evoluzione vera e propria del software in questione più efficientemente.

Bibliografia

[1] Luca Di Lorenzo: "*Project Automation*" - Progetto https://github.com/AsdInside/Automation_Example

[2] Porfirio Tramontana: *CONCURRENT VERSIONING SYSTEMS*. https://communitystudentiunina-my.sharepoint.com/:b:/g/personal/ptramont_unina_it/EV5e5HP77mdGnwWvWPIe3rcBe2sneHSQfHd2pekVoDy3zA

[3] Giorgio Fusari: "*Software testing, cos'è e come automatizzare il collaudo*", 28 Marzo 2022

[4] Vito Lavecchia: "*Che cos'è, importanza e vantaggi del test automatizzato del software*"

[5] "*Understanding GitHub Actions*" - https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

[6] "*Come si usa GIT e cosa é GitHub ?*" - https://vixr.it/guida-a-github-in-italiano-come-si-usa-git-e-cosa-e-github/

[7] "*What is Maven*" - https://maven.apache.org/what-is-maven.html#mavens-objectives , https://www.tutorialspoint.com/maven/maven_overview.htm

[8] "*Plug ins*" - https://www.tutorialspoint.com/maven/maven_plugins.htm

[9] "*Sviluppo Software*" - https://www.waveinformatica.com/news/wavepedia/guida-sviluppo-software/ , https://www.ibm.com/it-it/topics/software-development

[10] "*What is Automation Testing?*" - https://www.guru99.com/automation-testing.html

[11] "*How to build a CI/CD pipeline with GitHub Actions*" - https://github.blog/2022-02-02-build-ci-cd-pipeline-github-actions-four-steps/#:~:text=

[12] "*About packaging with GitHub Actions* " - https://docs.github.com/en/actions/publishing-packages/about-packaging-with-github-actions

[13] "*JUnit Tutorial*" - https://www.w3schools.blog/junit-tutorial

[14] "*maven-surefire-plugin*" - https://mvnrepository.com/search?q=maven-surefire-plugin

[15] "*Fork a repository*" - https://docs.github.com/en/get-started/quickstart/fork-a-repo

[16] "*Software Development Life Cycle (SDLC) Automation*" - https://linuxhint.com/software-development-life-cycle-automation/

[17] "*Automated Testing*" - https://smartbear.com/learn/automated-testing/what-is-automated-testing/

[18] "*Quickstart for GitHub Actions*" - https://docs.github.com/en/actions/quickstart

[19] "*YAML*" - https://learnxinyminutes.com/docs/yaml/ , https://www.redhat.com/it/topics/automation/what-is-yaml

[20] "*Building and testing Java with Maven*" - https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven

[21] "*About forks*" - https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/about-forks

Link repository Github del **lavoro di tesi**

- https://github.com/AlexKurukulasuriya/Example_Gestione

Ringraziamenti

Al termine di questo ciclo durato cinque anni, il primo sentito ringraziamento va al professor Porfirio Tramontana, che mi ha concesso l'onore di poter concludere assieme questo percorso di studi con un lavoro di cui ne ho fatto gelosamente tesoro. La sua disponibilità e lo scambio personale che ho avuto con lui è stato estremamente costruttivo, sotto ogni aspetto. A lui un grazie sincero.
Un ringraziamento speciale va alla famiglia Mirelli, senza la quali non sarei qui oggi. Spero che vi rendiate conto di quanto siete stati fondamentali per la mia famiglia. La vostra disponibilità e la generosità hanno dato inizio al mio percorso di studi in Italia e parte del merito di tutto questo è anche vostro. In un momento di tanta confusione e caos mi avete fatto sentire a casa ed è davvero un privilegio per me conoscere persone bellissime come voi. Grazie davvero di tutto.
Ringrazio il mio amico "incredibile" Carlo, anche se non ci sono sufficienti parole per ringraziarti. Sei una delle persone più importanti che abbia mai conosciuto e se sono qui è anche grazie a te. Da quando ci conosciamo, abbiamo fatto quasi tutto insieme e per me sei sempre stato una guida. Ogni singola scelta che ho preso in questi 9 anni, si è basata sul tuo esempio. Ci sei sempre stato per me e non avrei mai pensato di trovare un altro fratello così. Grazie per le tue parole, per il sostegno dal primo giorno e per la tua vicinanza. Non credo di essere all'altezza della tua amicizia ma ne sono tanto onorato. Grazie di cuore.
Vorrei ringraziare Tonino, Pupetta, Nello, Rosaria, Giusi, Antonio e Giovanna. Siete stati al mio fianco dal primo giorno e mi avete sempre sostenuto con le parole e le azioni. Ormai siete la mia famiglia e vi voglio troppo bene. Vi ringrazio di avermi accolto nelle vostre vite.
Alle sorelle Eleonora e Ida, non mi avete mai permesso di mollare, mi avete fatto capire quanto sia importante essere coraggiosi davanti alle sfide e impegnarsi per ottenere i risultati migliori. Mille grazie per la vostra guida e per la vostra presenza nella mia vita.

Così, si conclude una lunga avventura di studi, accompagnata da tante lacrime e voglie di mollare. Guardando indietro vedo solo un ragazzino che non conosceva neanche una parola in italiano e quindi dopo i primi mesi di scuola tornava a casa piangendo. Sono fiero di quello che sono riuscito a fare nonostante mille difficoltà. Ma non ce l'avrei

mai fatto senza mia mamma che mi ha sempre incoraggiato, con cui ho pianto nei momenti di debolezza e riso nei momenti di gioia. Mi ascolti e mi aiuti sempre a fare scelte migliori. Hai fatto dei sacrifici enormi per la famiglia e spero che questo mio piccolo traguardo ti abbia reso meritatamente felice. Mio padre, il quale ha sempre avuto il sogno di avere il primo laureato della famiglia, grazie per la tua saggezza e la determinazione che mi ha concesso di diventare la persona che sono oggi. Ringrazio mio fratello, che non è qui oggi, ma che mi ha sempre dato la forza di cui avevo disperatamente bisogno per portare a termine questo percorso. Mi chiedo spesso come sarebbe la mia vita oggi se le cose fossero andate diversamente. Laurearsi in ambito informatico è sempre stato uno dei tuoi sogni: Questo è per te! spero di averti reso orgoglioso, a presto.