Assignment 2 – DevOps

Group Name:

Gruppo MK

Group Members:

- Paolo Mascheroni Mat. 886220
- Kristian Kovacev Mat. 885839

Introduzione:

Questo documento è una relazione sul lavoro svolto nell'ambito del secondo Assignment, che riguarda la costruzione di una pipeline CI/CD che possa essere eseguita automaticamente dall'infrastruttura CI/CD di Gitlab. In particolare, questo documento riporta tutte le decisioni che sono state prese durante lo sviluppo del progetto e, per ogni stage della pipeline, fornisce una spiegazione del perché esse sono state prese.

Link al repository:

https://gitlab.com/gruppo-mk/2024_assignment2_Gruppo_MK.git

Applicazione utilizzata:

La scelta dell'applicazione utilizzata per testare la pipeline è ricaduta su un'app lato server, scritta utilizzando Node.js, che si incontra nel repository all'interno del file Magazzino.js, il cui funzionamento è descritto di seguito:

- L'app richiede di effettuare il login, inserendo username e password.
- Una volta effettuato il login, viene mostrato un menu a scelta multipla che permette di gestire ed eseguire alcune operazioni sui prodotti attualmente in magazzino, in particolare è possibile:
    1. Visualizzare (su prompt) tutti i prodotti di una certa categoria
    2. Cambiare il prezzo dei prodotti di una certa categoria.
    3. Visualizzare le informazioni di un prodotto.
    4. Cancellare un prodotto.
    5. Cancellare un'intera categoria di prodotti.
    6. Aggiungere un nuovo prodotto.
    7. Aggiungere una nuova categoria di prodotti.
    8. Effettuare il logout.
- Per autenticare il login, l'app si avvale di un file *Login.json* da cui legge le combinazioni username-password valide

- Per effettuare le operazioni sui prodotti, come la ricerca, l'aggiunta e la cancellazione viene utilizzato un file *Magazzino.json*, nel quale essi vengono memorizzati sotto forma di oggetti JSON.

Descrizione Pipeline:

Come da consegna, la Pipeline si compone di 6 stage: *build*, *verify*, *test*, *package*, *release* e *docs*.

```
stages:
  - build
  - verify
  - test
  - package
  - release
  - docs
```

A livello generale, prima dei job, viene dichiarata l'immagine Docker che dovrà essere utilizzata per tutti i job, cioè l'ultima versione stabile di Node.js (ad eccezione del job di release).

```
image: node:latest
```

Inoltre, viene creata una cache a livello globale, cioè accessibile da tutti i job, per salvare le dipendenze scaricate durante la fase di Build, in modo che i jobs successivi debbano preoccuparsi di installare solamente le dipendenze **npm** che necessitano per essere eseguiti, senza dover installare nuovamente le dipendenze del progetto.

```
cache:
  key: "$CI_COMMIT_REF_NAME" # Chiave unica per il commit
  paths:
    - node_modules/
```

La chiave **$CI_COMMIT_REF_NAME** garantisce che ogni branch o commit utilizzi una cache separata, evitando conflitti e garantendo un comportamento isolato.

Stages

1. **Build:** in questa fase, avviene l'installazione delle dipendenze utilizzate dal file di progetto, che sono specificate all'interno del file *package.json*. Durante la build, abbiamo deciso di utilizzare il comando

   ```
   npm install --only=production
   ```

   che installa solamente le dipendenze fondamentali all'esecuzione del progetto. Le " dev dependencies", utili ai job per eseguire i loro compiti, vengono installate singolarmente da ciascun job successivamente. In questo modo, non si rischia di installare prematuramente dipendenze utili a job che potrebbero non essere mai eseguiti a causa del fallimento di job precedenti.

```
compile:
  stage: build
  script:
    # Installiamo solo le dipendenze fondamentali per il progetto
    - npm install --only=production
```

2. **Verify:**
   - **Lint**: In questo job, il file viene analizzato staticamente utilizzando il package Eslint.

     ```
     lint:
       stage: verify
       script:
         - npm install eslint
         # Eseguiamo ESLint solo su Magazzino.js e generiamo un report in
     formato Json
         - npx eslint Magazzino.js -f json -o eslint-report.json
       # Ignoriamo il fallimento di Eslint per continuare con
       # la pipeline ed in seguito visualizzare i warning
       allow_failure: true
       # Gli artefatti salvati nei job precedenti non sono necessari
       dependencies: []
       # Salviamo il report di Eslint come artefatto
       artifacts:
         paths:
           - eslint-report.json
         when: on_failure
       # disabilitiamo la cache in questo job (non necessita le dipendenze di
     progetto)
       cache: {}
     ```

     Esso viene dapprima installato e poi eseguito, utilizzando il comando

     ```
     npx eslint Magazzino.js -f json -o eslint-report.json
     ```

     il quale esegue il linting solo sul file *Magazzino.js* (evitando di eseguirlo su tutti i file .js, come ad esempio, i file di test). Inoltre, salva il report di Eslint in formato JSON nel file *eslint-report.json*. Questo file viene poi salvato come artefatto.

     La cache viene dichiarata disabilitata, siccome questo job non necessita le dipendenze di progetto per funzionare (essendo analisi statica del codice). Vengono inoltre disabilitate le dipendenze, per non importare artefatti prodotti dagli altri job inutilmente.

   - **Flow:** In questo job, si utilizza Flow per effettuare un'analisi del file.

     ```
     flow:
       stage: verify
       script:
         - npm install flow-bin
         # Eseguiamo Flow solo su Magazzino.js e salviamo il report in formato
     JSON
         - npx flow check Magazzino.js --json > flow-report.json
     ```

```
  # Ignoriamo il fallimento di Flow per continuare con la
  # pipeline ed in seguito visualizzare i warning
  allow_failure: true
  # Gli artefatti salvati nei job precedenti non sono necessari
  dependencies: []
  # Salviamo il report JSON come artefatto
  artifacts:
    paths:
      - flow-report.json
    when: on_failure
    # Impediamo al job di scrivere sulla cache in quanto
    # il package flow-bin non serve agli altri job
  cache:
    key: "$CI_COMMIT_REF_NAME"
    paths:
      - node_modules/
    policy: pull
```

Dopo essere stato installato, il package flow viene utilizzato sul file Magazzino.js. Come nel job di Lint, vengono salvati il report prodotto come artefatto nel file *flow-report.json*. In questo caso, la cache viene letta ma non scritta dal job. In questo modo, si possono utilizzare le dipendenze di progetto senza sovrascrivere la cache
con dipendenze quali *FLOW-BIN* che non sono utilizzate da nessun job successivo.

3. **Test:**
    1. **Unit-test** in questo job, si esegue il testing del file Magazzino.js.

```
unit-test:
  stage: test
  # installiamo jest e dei plug-in per permettere il salvataggio dei
report
  script:
    - npm install jest
    - npm install --save-dev jest-json-reporter
    - npm install --save-dev jest-junit
    # eseguiamo i test sul file Magazzino.js e salviamo i risultati in
formato .json
    - npm test __test__/Magazzino.test.js -- --json --
outputFile=reports/test-results.json
  # Facciamo override della cache globale e salviamo le
  # dipendenze jest scaricate in una seconda cache dedicata ai test
  cache:
    key: "$CI_COMMIT_REF_NAME-tests"
    paths:
      - node_modules/
    policy: push
  # Gli artefatti salvati nei job precedenti non sono necessari
  dependencies: []
  # Salviamo i risultati nella cartella reports come artefatto
  artifacts:
    paths:
      - reports/test-results.json   # formato .JSON
      - reports/junit/junit.xml     # formato .xml
```

```
    when: on_failure
```

I test sono scritti all'interno del file *__test__/magazzino.test.js* e testano le principali funzioni del file di progetto *Magazzino.js*, considerando sia casi di successo sia casi di fallimento delle stesse. Alcune delle funzioni testate sono:

- caricaMagazzino
- controlloLogin
- trovaIDCategoria
- stampaProdotti
- cambioPrezzo
- cercaProdotto
- cancellaProdotto
- cancellaCategoria
- aggiungiProdotto
- aggiungiCategoria
- nomiCategoria
- nomiProdotti

Il package utilizzato per effettuare i test è ***Jest***, il quale viene installato per eseguire il job ma non viene salvato sulla cache (grazie al comando *policy:pull*). Viene definita una nuova cache, effettuando l'override di quella globale, per garantire che il job successivo, l'integration-test, non debba scaricare nuovamente le dipendenze Jest. Non necessitiamo di artefatti prodotti da altri jobs.

2. **Integration-test:** In questo job vengono eseguiti i test d'integrazione sull'applicazione definiti nel file *__test__/integration_test.test.js.* Essi sono eseguiti sempre tramite jest.

```
integration-test:
  stage: test
  # Facciamo in modo che integration-test venga eseguito
  # solo in caso di successo di unit-test
  needs:
    - unit-test
  script:
    # eseguiamo i test sul file Magazzino.js e
    # salviamo i risultati in formato .json
    - npm test __test__/integration_test.test.js -- --json --
outputFile=reports/integration-results.json
  # Leggiamo la cache specifica dei test
  cache:
    key: "$CI_COMMIT_REF_NAME-tests"
    paths:
      - node_modules/
    policy: pull
  # Gli artefatti salvati nei job precedenti non sono necessari
  dependencies: []
  # Salviamo i risultati nella cartella reports come artefatto
  artifacts:
```

```
    paths:
      - reports/integration-results.json     # formato .JSON
      - reports/junit/junit.xml              # formato .xml
    when: on_failure
```

Con il comando

```
needs: - unit-test
```

definiamo una dipendenza logica tra il job di test unitario e quello di integrazione. In questo modo, non eseguiamo mai il test d'integrazione se prima i test unitari non hanno avuto successo.
Questo è utile considerando che i test d'integrazione possono richiedere molto più tempo dei test unitari.
Leggiamo dalla cache definita nel job di test unitario e non la modifichiamo, grazie al comando *policy: pull*.
I test d'integrazione eseguiti sono i seguenti:

- Scenari di login e visualizzazione prodotti di una categoria
- Scenari di aggiunta e cancellazione prodotti
- Scenari di aggiunta e cancellazione categorie e prodotti associati
- Scenari di login e visualizzazione di un prodotto
- Scenari di login e cambio prezzo a dei prodotti

Quello che viene testato in questo job sono le due funzioni *login()* e *cli()*, le quali necessitano di un'interazione da linea di comando e per come sono definite sono le più indicate per eseguire test d'integrazione, siccome sono loro a chiamare tutte le altre funzioni, in base all'operazione che l'utente vuole effettuare.

4. **Package**: in questo stage, creiamo il package del progetto. In particolare, esso viene creato tramite lo strumento ***Webpack***, che viene installato ed utilizzato solo in questo job. Webpack necessita di un file di configurazione per poter funzionare chiamato *webpack.config.js*. Grazie a questo file, quando viene eseguito lo script di build con
npm run build, Webpack sa in che modo deve essere creato il package. Nel nostro caso, Webpack integra in un solo file *bundle.js* il codice del progetto e tutte le dipendenze che esso necessita, applica il transpiling del file grazie a Babel e abilita la minificazione del codice. Quindi, genera una cartella */dist* che contiene il file *bundle.js* e i file *Login.json* e *Magazzino.json*. Questo package contiene il codice del progetto e tutte le dipendenze e,d in generale, tutto ciò che necessita per essere eseguito.

```
webpack:
  stage: package
  script:
  # Installiamo webpack per creare il package
    - npm install --save-dev webpack webpack-cli
     # Installiamo Babel per utilizzarlo nel file webpack.config.js
    - npm install --save-dev babel-loader @babel/core @babel/preset-env
    - npm install copy-webpack-plugin --save-dev
    # Eseguiamo lo script di build definito in Package.json
    - npm run build
  # Gli artefatti salvati nei job precedenti non sono necessari
```

```
  dependencies: []
  # Salviamo il bundle nella directory dist/ come artefatto
  artifacts:
    paths:
      - dist/
    when: on_success
  cache:
    key: "$CI_COMMIT_REF_NAME"
    paths:
      - node_modules/
    policy: pull
```

Dopodiché, salviamo il package prodotto nella cartella */dist* come artefatto (solo quando il job ha successo).

5. **Docker-image:** In questo stage creiamo l'immagine Docker del pacchetto appena creato nello stage di package. Per fare ciò, utilizziamo il servizio *Docker-in-Docker* che ci permette di eseguire comandi Docker nel container.

```
docker-image:
  stage: release
  # utilizziamo l'ultima versione di docker stabile
  image: docker:stable
  services:
    - docker:dind
  # disabilitiamo la cache in questo job (non necessita le dipendenze di
progetto)
  cache: {}
  # dichiariamo la dipendenza dal job webpack per poter utilizzare
l'artefatto /dist
  dependencies:
    - webpack
  script:
    # login al Container registry
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
    # scarichiamo l'immagine se già esiste
    - docker pull $CONTAINER_IMAGE:$CI_COMMIT_REF_NAME || true
    # costruiamo la nuova immagine (sfruttando la cache di quella precedente
se esiste)
    - docker build --cache-from $CONTAINER_IMAGE:$CI_COMMIT_REF_NAME -t
$CONTAINER_IMAGE:$CI_COMMIT_REF_NAME .
    # carichiamo la nuova immagine sul Container Registry con lo stesso nome
della precedente
    - docker push $CONTAINER_IMAGE:$CI_COMMIT_REF_NAME
```

Non necessitiamo di librerie esterne per questo job quindi disabilitiamo la cache. Aggiungiamo la reference al job precedente (webpack) per poter accedere alla cartella */dist*, dove si trova il package. I comandi dello script servono per eseguire il login a Container Registry, scaricare l'immagine dal registro, costruire la nuova immagine utilizzando la vecchia come cache ed infine caricare la nuova immagine nel Container Registry.

Per costruire l'immagine utilizziamo un *Dockerfile* così composto:

```
# Usa un'immagine base Node.js
FROM node:latest
```

```
# Imposta la directory di lavoro
WORKDIR /app

# Copia i file generati nella directory dist
COPY dist/ /app/

# Espone la porta 3000
EXPOSE 3000

# Comando di avvio
CMD ["node", "bundle.js"]
```

Prima di tutto carichiamo un'immagine di node (l'ultima versione stabile). Successivamente, copiamo il package */dist* nella cartella */app* e impostiamo */app* come cartella di lavoro. Esponiamo la porta 3000, per consentire la comunicazione tra il container e l'esterno. Infine, impostiamo come comando di avvio node *bundle.js*.

6. **Docs**: In questo stage utilizziamo la libreria *documentation.js* per generare la documentazione del progetto ed in seguito pubblicarla su Git Lab Pages.

```
pages:
  stage: docs
  script:
    - npm install --save-dev documentation
    - npm run docs
  # Gli artefatti salvati nei job precedenti non sono necessari
  dependencies: []
  artifacts:
    paths:
      - public
  # disabilitiamo la cache in questo job (non necessita le dipendenze di
progetto)
  cache: {}
```

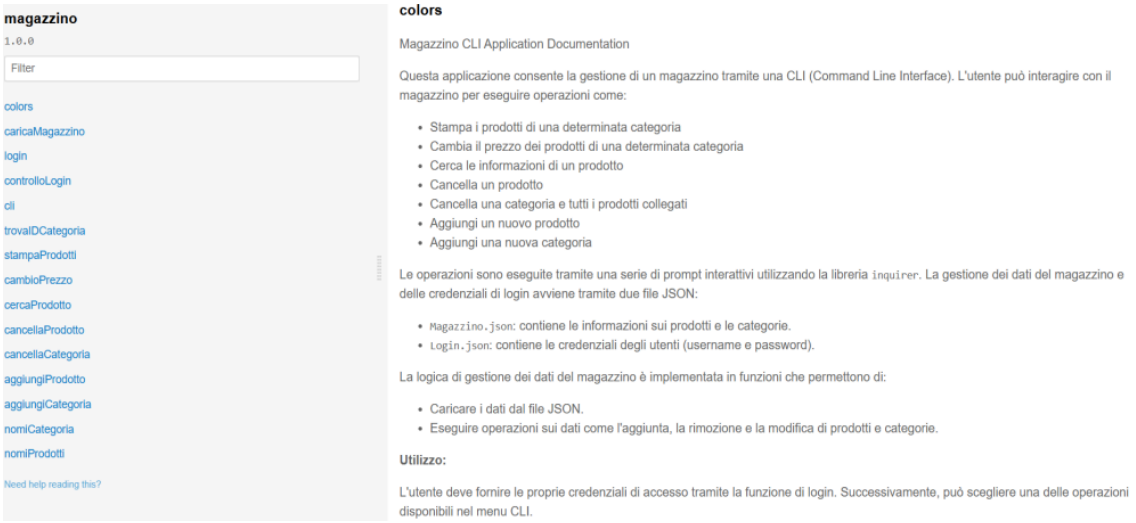
Per fare questo, scarichiamo la *documentation.js* ed eseguiamo lo script con il comando *run docs*. Lo script chiama la funzione documentation build sul file *Magazzino.js* e salva il file creato in formato .html in una cartella *public*.

Questa funzione genera la documentazione leggendo i commenti in JSDoc inseriti nel testo, come questo:

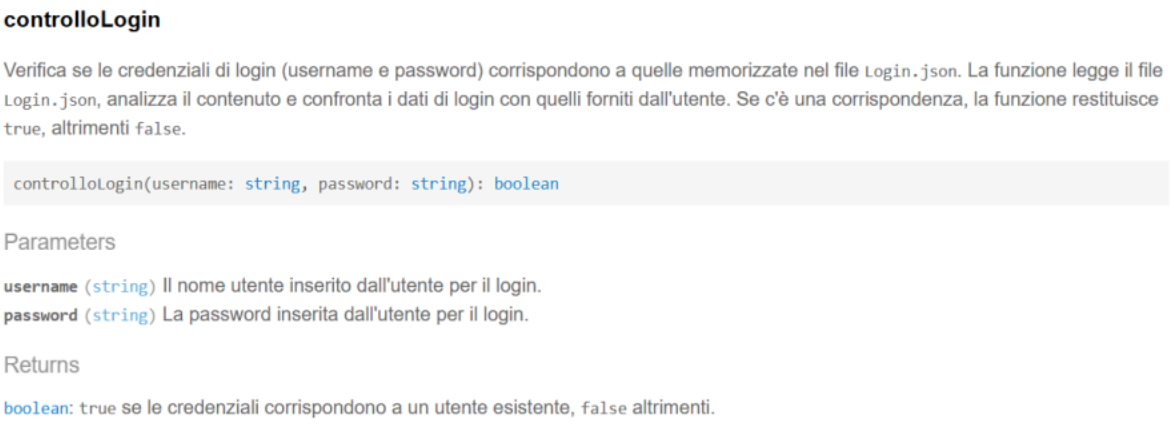
```
/**
 * Carica i dati del magazzino dal file `Magazzino.json`.
 * Questa funzione legge il file JSON, lo analizza e restituisce i dati del
magazzino come oggetto JavaScript.
 *
 * @function
 * @returns {Object} Restituisce l'oggetto `magazzino` che contiene i dati
dei prodotti e delle categorie.
 */

function caricaMagazzino() {
    const contents = fs.readFileSync("Magazzino.json");
    magazzino = JSON.parse(contents);
    return magazzino;
}
```

In questo modo, si ottiene una pagina *index.html* con questa forma:



In cui ogni funzione è descritta e commentata, ad esempio:



Infine, salviamo la cartella *public* come artefatto e disabilitiamo la cache, siccome questo job necessita solo la dipendenza *documentation.js* per funzionare correttamente. Successivamente, questo documento è caricato sulle GitLab Pages.