

Indice

Obbiettivo dell'elaborato	1
Analisi.....	1
Dominio	2
Requisiti non funzionali	2
Requisiti funzionali.....	2
Progettazione.....	3
Modello di dominio	4
Business Logic	7
Deployment	9
Configurazioni.....	15
Test.....	16
Front-end docker.....	21
RXJS.....	22
Sitografia	22

Obbiettivo dell'elaborato

Il progetto prevede lo sviluppo di un applicativo web che permetta la registrazione, autenticazione e relativa gestione di utenti.

Per realizzare tale scenario la tecnologia principalmente utilizzata è stata "Spring".

Inoltre, sono state utilizzate tecnologie quali: Angular, Docker e Jenkins. La prima è stata utilizzata per realizzare il livello di presentazione, andando a implementare la vista e l'interazione dell'utente con questa, oltre che la gestione delle REST-API per lo scambio di dati tra Front-end e Back.end. Docker e Jenkins svolgono principalmente un ruolo non funzionale, implementando servizi di continua integrazione e distribuzione. Infine, sono stati utilizzati due diversi DBMS: H2, MySQL. Il primo utile in fase di testing, mentre il secondo è stato utilizzato in una fase più avanzata del progetto, grazie ai servizi di persistenza più avanzati che mette a disposizione.

Analisi

Nel capitolo che segue verranno presentati i principali modelli concettuali utili ad avere una visione generale del sistema. Vengono di seguito esposti i requisiti funzionali e non funzionali, oltre che una descrizione del dominio di applicazione. Tali informazioni verranno poi concretizzate nella fase di progettazione.

Dominio

Il dominio che interessa il sistema è quello di una generica azienda dove le entità principali sono gli utenti, le risorse, i ruoli, le operazioni e i vari permessi.

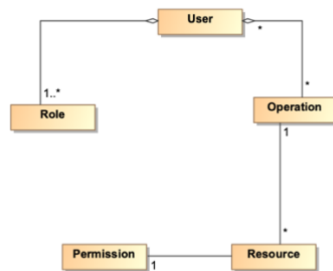


Figura 1: Diagramma delle classi

Requisiti non funzionali

Dal punto di vista non funzionale è previsto lo sviluppo di due applicativi separati: Front-end e Back-end. Ciò risulta importante nell'ambito di un'architettura che si basa sul paradigma Rest. Il Front-end si appoggia all'utilizzo del web framework Angular 10.

Altro punto importante riguardo i requisiti non funzionali è importante evidenziare la struttura del deployment e della sicurezza.

La fase di deployment è stata realizzata attraverso l'uso di Jenkins, Docker, Git. La parte di sicurezza utilizza come supporto il Framework Spring-Security.

Requisiti funzionali

Per spiegare i requisiti funzionali vengono di seguito esposti i vari Mockups del sistema.

L'interfaccia del sistema si basa su un paradigma a eventi, dove l'azione dell'utente scatena una serie di eventi.

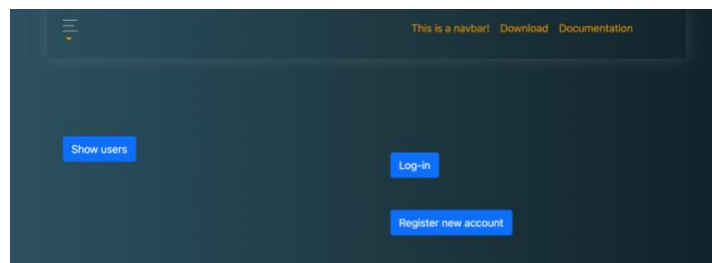


Figura 2

Nell'immagine si osserva l'intera interfaccia grafica. I tre pulsanti corrispondono ai tre casi d'uso che l'utente può consumare.

Il primo (Show users) permette di reperire tutti gli utenti presenti nel database con la rispettiva richiesta (GET-REQUEST). Nell'immagine 3 si può osservare l'output generato una volta cliccato il bottone.

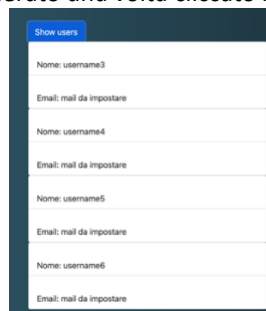


Figura 3

Il bottone Log-in permette di verificare la presenza di un utente nel sistema back-end attraverso la GET-REQUEST.

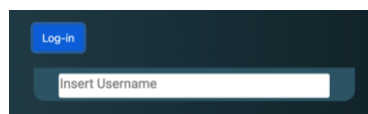


Figura 4: Input dati

Infine, il caso d'uso "Register new account" permette attraverso una POST_REQUEST di inserire un nuovo utente nel database presente nel lato back-end.

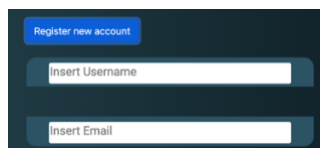


Figura 5: Show users output

Progettazione

Per quanto riguarda l'architettura del sistema si osserva di seguito il "Package Diagram" del sistema.

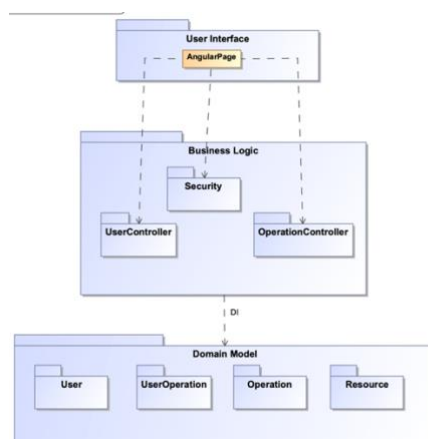


Figura 6: architettura generale del sistema

Modello di dominio

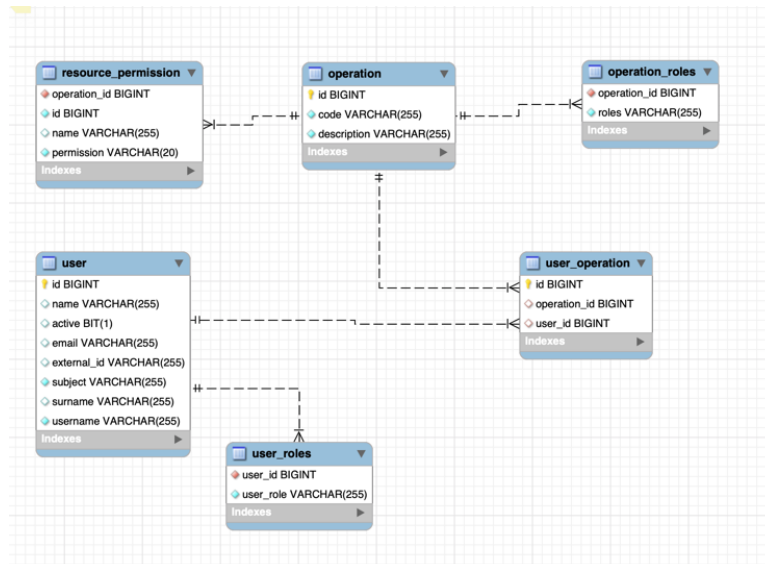


Figura 7: Entity-Relationship Diagram

Come si osserva nel diagramma entità-relazione, le entità sono: user, operation, user_operation e resource_permission. Inoltre, è possibile osservare le varie associazioni. In particolare, la prima relazione da sottolineare è quella tra utente e ruolo.

Inoltre, viene riportato il diagramma delle classi in prospettiva di implementazione nella figura 8:

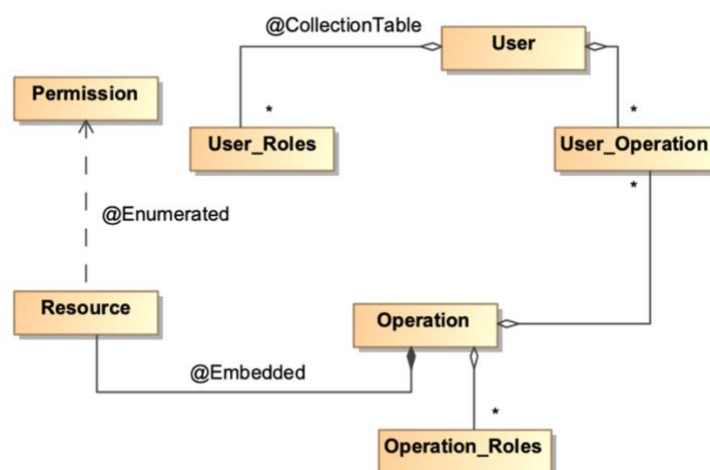


Figura 8: Diagramma classi in prospettiva di progettazione

La relazione tra user e ruoli è realizzata attraverso una “ElementCollection”, posta come annotazione sopra l’attributo lista di stringhe (Figura 8).

```

@ElementCollection
@CollectionTable(name = "user_roles", joinColumns = @JoinColumn(name = "user_id", nullable = false))
@Column(name = "user_role", nullable = false)
private List<String> roles = new ArrayList<>();

```

Figura 9: ElementCollection di un tipo semplice (String)

La tabella user_operation serve a mettere in relazione gli utenti con le operazioni. Per fare ciò vi è una relazione uno a molti sia tra user che tra operation nei confronti dell'entità user_operation. Un esempio si vede nell'immagine sotto:

```

@OneToMany(mappedBy = "user", fetch = FetchType.LAZY, cascade = CascadeType.ALL)
private List<UserOperation> userOperation = new ArrayList<>();

```

Figura 10: relazione di molteplicità oneToMany tra User e UserOperation

L'entità "operation" mette in relazione i ruoli con le risorse. Infatti, a ogni operazione sono associati più ruoli e più risorse. Infine, la classe ResourcePermission è annotata come Embeddable. Grazie a questa annotazione, l'entità Operation può rappresentare una ElementCollection di un tipo non semplice, come si osserva nel codice in basso.

```

@ElementCollection
@CollectionTable(name="Resource_Permission")
private List<ResourcePermission> resources = new ArrayList<>();

```

Figura 11: ElementCollection di entità Embeddable

La classe "ResourcePermission" specifica gli attributi: id, nome (url della risorsa), permission. L'ultimo attributo è un enumerativo con tre valori:

```

public enum PermissionEnum {

    READ( value: "READ"),
    WRITE( value: "WRITE"),
    AUTHORIZE( value: "AUTHORIZE");
}

```

Figura 12

Il modello di dominio contiene l'insieme di servizi che gestiscono le entità e la manipolazione di queste. Nella figura numero 7 è mostrato il package diagram relativo al modello di dominio.

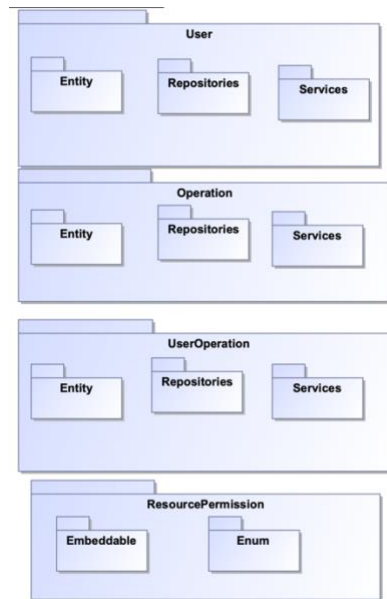


Figura 13: architettura modello di dominio

Ogni entità è gestita da una rispettiva classe Repository (che si trova dentro il package “Repositories”), inoltre all’interno del pacchetto Services vi sono le varie classi: i.e UserService, OperationService, ecc. Il modulo Repository implementa il pattern repository.

Come esempio è stato riportato il diagramma delle classi che descrive l’interazione tra l’entità e i servizi messi a disposizione dall’interfaccia Repository e la classe Service nel caso del pacchetto Operation (Figura 13).

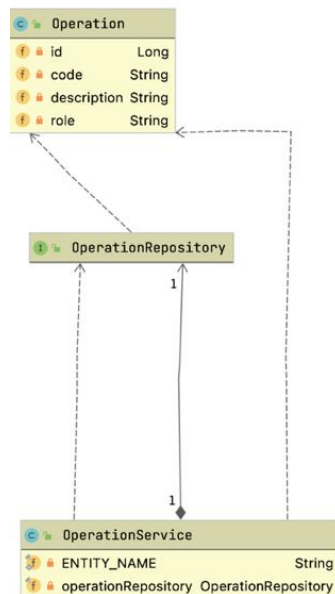


Figura 14: diagramma classi interazione entità con relativo repository e service.

Nel caso specifico del progetto l’interfaccia che implementa “Repository” è “OperationRepository”. Infine, vi è la classe “OperationService”. Tale classe è annotata con “@Service”. Questa annotazione è un’estensione di “@Component” e come tale eredita le sue funzionalità. Come verrà spiegato meglio in seguito, nella classe OperationServiceFacade viene iniettato il servizio di “OperationService”.

Business Logic

Il package Business Logic contiene il codice bootstrap del programma, un pacchetto Security, uno UserController e uno ApplicationController.

Nel pacchetto Security sono presenti le logiche di sicurezza che fanno uso delle dipendenze di Spring Security.

In particolare, tale pacchetto implementa la funzionalità di filtro delle richieste. Permette di specificare (in formato stringa o pattern di stringhe) un insieme di end-point pubblici o privati. Inoltre, analizzando l'header delle richieste vi è la possibilità di autenticare eventuali utenti.

In questa maniera ogni qual volta giungono richieste a un end-point privato queste restituiranno il codice di errore 403, a seconda che la richiesta sia legittima o meno.

```
@Override
protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain)
    throws IOException, ServletException {

    String headerXauthSubject = req.getHeader( s: "x-auth-subject");
    String headerXauthRole = req.getHeader( s: "x-auth-roles");
    String eMail = req.getHeader( s: "x-auth-email");
    String username = req.getHeader( s: "x-auth-username");
    String userid = req.getHeader( s: "x-auth-userid");
    logger.debug("x-auth-subject {}", headerXauthSubject);
    logger.debug("x-auth-roles {}", headerXauthRole);

    UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(headerXauthSubject, credentials: null, authorities);
    SecurityContextHolder.getContext().setAuthentication(authentication);
    logger.debug(String.valueOf(SecurityContextHolder.getContext().getAuthentication().isAuthenticated()));
    chain.doFilter(req, res);
}
```

Figura 15: Codice autenticazione (Spring Security)

Nell'immagine in basso è presente l'insieme delle classi che gestiscono un end-point:

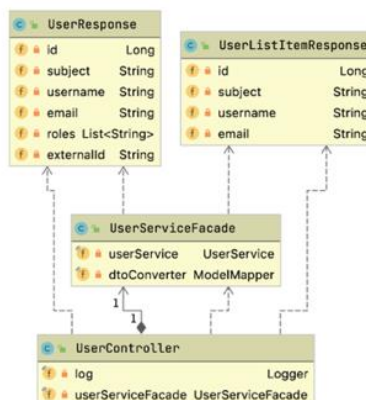


Figura 16: Diagramma classe autogenerato

Come si osserva nell'esempio della figura 17, gli end-point sono composti da tre tipologie di classi:

1. Facade: Implementa il pattern Facade, facendo sì che la classe controller si interfacci solo con la classe Facade e che sia questa a gestire le funzionalità di DTO e gestione delle entità.
2. Response: Rappresenta l'oggetto su cui sarà chiamata la funzione dtoConverter (Response, User).
3. Controller: Implementa i metodi CRUD con le relative configurazioni per rendere disponibili i vari end-point.

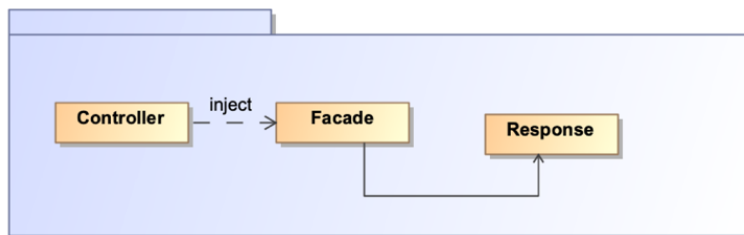


Figura 17: Struttura generica end-point

La classe Controller espone l'insieme degli end-point in stile CRUD. Di seguito un esempio di end-point:

```

@CrossOrigin(origins = "http://localhost:4200")
@ApiOperation(value = "Retrieve user", notes = "Retrieve user", response = User.class)
@ApiResponses({
    @ApiResponse(code = 200, message = "operation retrieved", response = User.class),
})
@GetMapping(value = "/{idUser}", produces = MediaType.APPLICATION_JSON_VALUE)
public UserResponse getUser(@PathVariable("idUser") Long idUser) {
    log.info(" method start with id: {}", idUser);
    UserResponse user = this.userServiceFacade.getUserById(idUser);
    log.info("getCallCenter method end with id: {}", idUser);
    return user;
}

```

Figura 18: annotazioni Swagger,annotazioni Mapping

Come si osserva nell'immagine 18, gli end-point sono decorati da alcune notazioni:

1. -Swagger Annotation: Swagger è un Framework esterno che permette di generare e testare molte informazioni utili riguardo l'end-point. In particolare, quando gli end-point vengono annotati con le Swagger-annotation, queste informazioni possono essere visionate usando il client <http://localhost:8080/your-app-root/swagger-ui.html>.
2. -Mapping Annotation: permette di specificare il tipo di end-point (get, post, put, delete) e il path (value), oltre che altre eventuali informazioni .

Il body della funzione end-point getUser processa l'id passato come argomento e attraverso la classe Facade genera la risposta:

Deployment

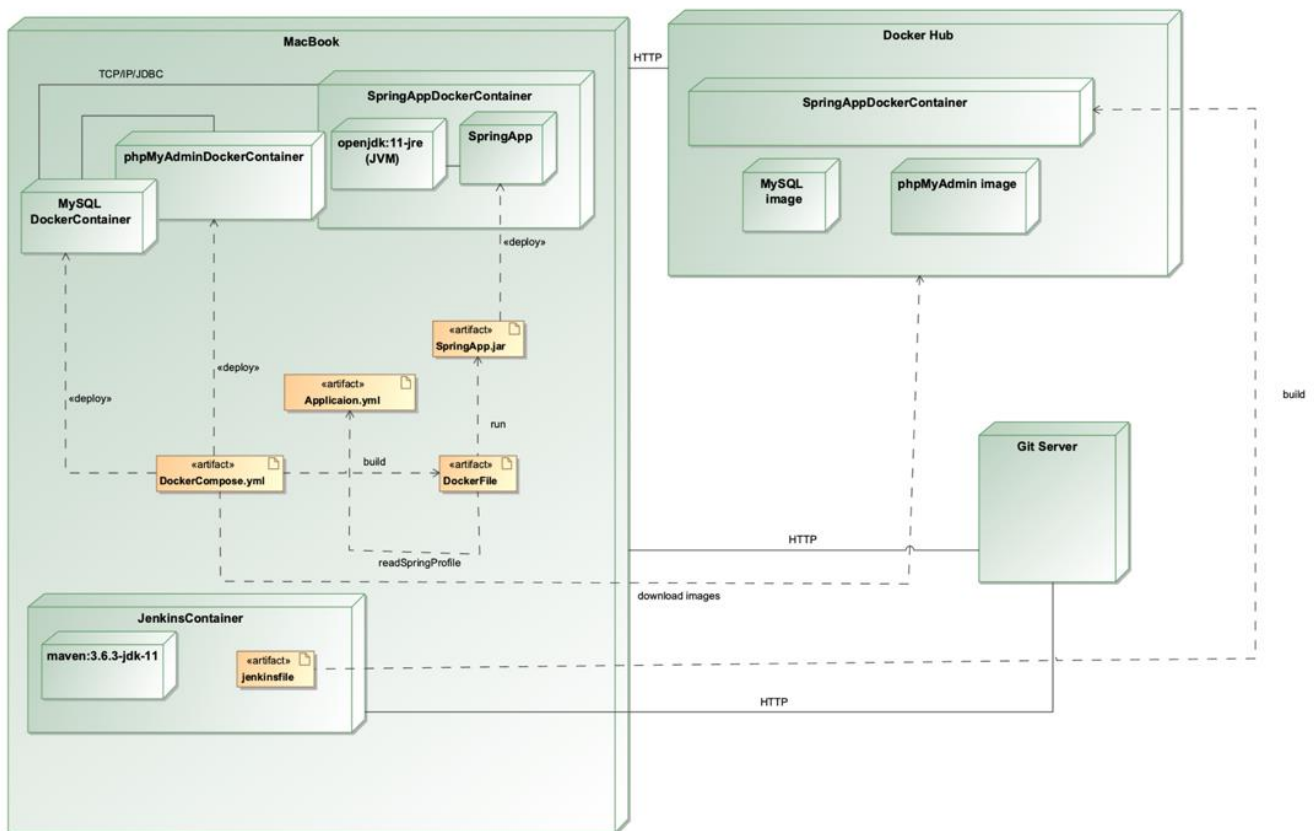


Figura 19: Deployment Diagram

Nel diagramma di distribuzione si osserva che i principali nodi fisici siano 3: MacBook, Git Server, Docker Hub. Questi tre nodi si trovano in locazioni geografiche differenti. Per comunicare tra di loro utilizzano il protocollo HTTP o il più sicuro HTTPS.

All'interno del nodo principale (MacBook) risiedono tre container ognuno dipendente dall'altro.

Il container principale (SpringAppDockerContainer), come si osserva nella figura 23, viene avviato partendo da un'immagine di una Java Virtual Machine, i quali servizi vengono utilizzati per avviare il programma ".jar".

Tutte le istruzioni sono specificate nel DockerFile. Ultima nota riguarda il profilo Spring utile per l'esecuzione del jar file. Le informazioni del profilo vengono lette nel file Application.yml. In particolare, vengono lette caratteristiche quali:

1. -L'utilizzo della sicurezza.
2. -Il database sul quale i servizi dell'applicazione agiranno (URL, configurazioni utili, username e password).
3. -L'utilizzo del Framework di riferimento per la gestione dei processi DDL e DML dei dati (Hibernate o Liquibase).
4. -Base Context.
5. -Altri parametri utili all'app.

Nella figura seguente è riportata la struttura del DockerFile:

```

# Start with a base image containing Java runtime
FROM openjdk:11-jre

## Add a volume pointing to /tmp
#VOLUME /tmp

# Make port 8080 available to the world outside this container
EXPOSE 8080

# The application's jar file
ARG JAR_FILE=business-logic-layer/target/operations-backend.jar

# Add the application's jar to the container
ADD ${JAR_FILE} operations-backend.jar

# Run the jar file
#ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/operations-backend.jar"]

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-Dspring.profiles.active=test1","-jar","/operations-backend.jar"]

```

Figura 20: DockerFile

Il file DockerCompose.yml svolge un ruolo importante nel coordinare la creazione dei tre container e nel gestire l'interazione tra queste.

Tale file, come si osserva nell'immagine sotto, utilizza il DockerFile che si occuperà di creare il container apposito della spring-app.

```

#service 3: definition of your spring-boot app
customerservice:
    #it is just a name, which will be used only in this file.
    image: customer-service
    #name of the image after dockerfile executes
    container_name: customer-service-app
    #name of the container created from docker image
    build:
        context: .
        #docker file path (. means root directory)
        dockerfile: Dockerfile
        #docker file name
    ports:
        - "8080:8080"
        #docker container port with your os port
    restart: always

    depends_on:
        #define dependencies of this app
        - db
        #dependency name (which is defined with this name 'db' in this f

    environment:
        SPRING_DATASOURCE_URL: jdbc:mysql://mysql-db2:3306/operations?createDatabaseIfNotExist=true
        SPRING_DATASOURCE_USERNAME: root
        SPRING_DATASOURCE_PASSWORD: mukit

```

Figura 21: DockerCompose file

Ci sono inoltre altri 2 servizi presenti nel DockerCompose.yml, questi hanno lo scopo di creare il servizio container MySQL e phpMyAdmin:

```

version: '3.3'
services:
  #service 1: definition of mysql database
  db:
    image: mysql:latest
    container_name: mysql-db2
    environment:
      - MYSQL_ROOT_PASSWORD=mukit
      - MYSQL_USER=root
    ports:
      - "3306:3306"
    restart: always

  #service 2: definition of phpMyAdmin
  phpmyadmin:
    image: phpmyadmin/phpmyadmin:latest
    container_name: my-php-myadmin
    ports:
      - "8082:80"
    restart: always

    depends_on:
      - db
    environment:
      SPRING_DATASOURCE_USERNAME: root
      SPRING_DATASOURCE_PASSWORD: mukit

```

Figura 22: DockerCompose file

Si osserva che l'istruzione che esplicita il collegamento tra la spring-app e il DB MySQL si riscontri nel file Application.yml nell'immagine sotto.

```

spring:
  profiles: test1
  jpa:
    properties:
      hibernate:
        jdbc:
          time_zone: CET
          dialect: org.hibernate.dialect.MySQL5InnoDBDialect
      hibernate:
        ddl-auto: none
    datasource:
      url: jdbc:mysql://mysql-db2:3306/customer?createDatabaseIfNotExist=true
      username: operations
      password: operations

```

Figura 23: Application.yml, profilo test2

Nel nodo MacBook è presente un altro container: JenkinsContainer.

Tale container svolge il ruolo di automatizzare la fase building di tutti i servizi prima spiegati, oltre che posizionare la spring-app sul docker hub (remoto).

Viene ora spiegata la struttura e i servizi offerti dal sistema Jenkins.

Dopo una fase preliminare di installazione di Jenkins all'interno di un container con il relativo port mapping (utile ad accedere all'interfaccia grafica), l'installazione dell'ultima versione di xcode (Mac) è possibile configurare lo strumento.

La prima funzionalità che mette a disposizione Jenkins è quella di collegarsi a un end-point Git.

In questa maniera è possibile usufruire di molti vantaggi messi a disposizione dai servizi offerti di Git. Oltre a ciò, è possibile configurare il server Git in maniera tale che si metti in ascolto di eventi.

In particolare, è possibile mettere in ascolto il server Git di un evento "push" da parte di un eventuale developer. In risposta a tale evento Git invierà una notifica al server Jenkins.

Tale notifica scatenerà la build di Jenkins.

L'immagine sotto riassume quanto detto in precedenza :

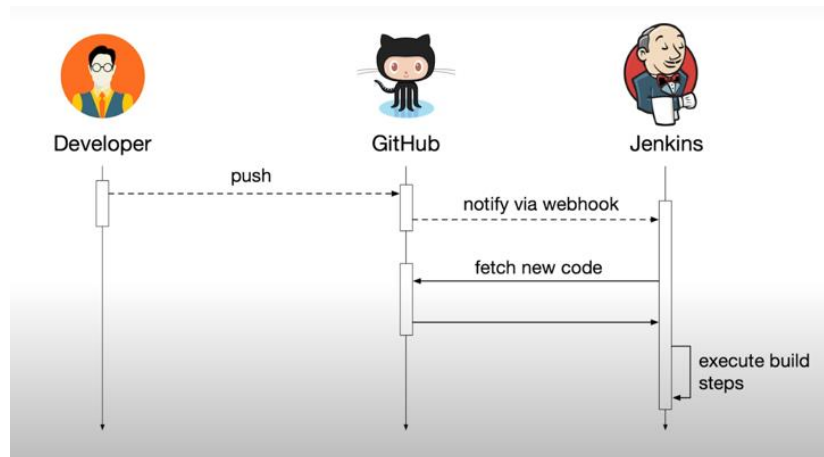


Figura 24: Sequenza build Jenkins-GitHub

La pipeline che viene eseguita ogni volta che un developer attua un push sul Git remoto è quella della figura 25.

```

pipeline {
  environment {
    registry = "niccocorsa/it-corsani"
  }
  agent {
    docker{
      image "maven:3.6.3-jdk-11"
    }
  }
  stages {
    stage('Image') {
      steps {
        script {
          image = docker.build registry + ":1"

          docker.withRegistry('', 'credentialjenkins') { ///credentialjenkins è la credenziale inserita su Manage J
            image.push("latest")
          }
        }
      }
    }
  }

  post {
    failure {
      echo 'Failure'
    }
    fixed {
      echo 'Fixed'
    }
  }
}

```

Figura 25: Pipeline Jenkins

Come si osserva nel codice mostrato sopra vi è l'utilizzo di vari servizi quali:

- Bash
- Docker
- Maven-jdk

I servizi bash si concretizzano ad esempio nell'uso di codice quale "echo 'failure'" che stampa nel terminale Jenkins 'Failure' quando si genera un'eccezione o errore.

Inoltre, Jenkins mette a disposizione un terminale che permette l'esecuzione di comandi in maniera molto simile a quanto accade per il sistema operativo Linux.

Il servizio Maven-jdk permette all'istanza di Jenkins di eseguire comandi Java (e.g java ex.jar) e Maven (e.g mvn clean install).

Il servizio di Docker richiede una configurazione particolare. Infatti, la macchina virtuale dove risiede Jenkins è un container creato con Docker. Per tale motivo è importante installare i servizi messi a disposizione da Docker all'interno del container di Jenkins. Per fare ciò è necessario accedere alla bash di Jenkins eseguire il comando "apt-get update && apt-get install -y docker.io".

Una volta installati i servizi Docker vi sono quattro ulteriori passaggi da fare:

1. Creare un volume in condivisione tra macchina ospitante e Docker-Container.
2. Concedere i privilegi necessari a comunicare (chmod 777 /var/run/docker.sock).
3. Creare una socket di comunicazione con i servizi Docker presenti sulla macchina ospitante e il container.
4. Gestire le credenziali con un Plug-in di Jenkins apposito.

Altre informazioni nel file istruzioni-jenkins al repository GitHub.

Attraverso i servizi di Docker installati è ora possibile scaricare velocemente immagini dal Docker hub ogni volta che vi è la necessità (come nel caso dell'immagine maven:3.6.3-jdk-11). Oltre a ciò, è possibile caricare sul repository remoto (Docker Hub) l'immagine della spring-app.

Oltre che l'environment di CI su Jenkins di seguito sono riportati i passi per attuare la CI su github actions. Una volta recato nella home-page di sonarcloud e aver selezionato la funzionalità di sonar su github sono state seguite le istruzioni per creare l'environment per attuare la CI con github e github actions. Per poter usufruire della versione di sonarcloud free è necessario che non vi siano repository privati su github. Per ragioni di sicurezza sarà necessario impostare un "segreto" come nell'immagine in basso.

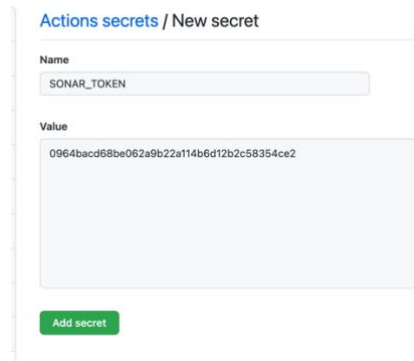
The image shows a web interface for creating a new GitHub Actions secret. At the top, it says "Actions secrets / New secret". Below this, there is a form with two fields: "Name" and "Value". The "Name" field contains the text "SONAR_TOKEN". The "Value" field contains a long alphanumeric string: "0964bacd68be062a9b22a114b6d12b2c58354ce2". At the bottom of the form, there is a green button labeled "Add secret".

Figura 26

Una volta impostato il segreto è possibile creare il file maven-publish.yml, che sarà posto nella cartella ".github/workflows/". Il sistema di github action riconoscerà automaticamente la presenza di un file contenente una pipeline di CI. Aggiungendo la proprietà al file pom.xml che si osserva nell'immagine in basso, si scatenerà la revisione di sonarcloud.

```
<properties>
  <sonar.organization>niccolocorsani</sonar.organization>
  <sonar.host.url>https://sonarcloud.io</sonar.host.url>
</properties>
```

Figura 27

La pipeline verrà eseguita ogni volta che vi sarà un push sul branch "main", come si osserva di seguito:

```
on:
  push:
    branches:
      - main
```

Figura 28

Di seguito sono osservabili i risultati della pipeline:

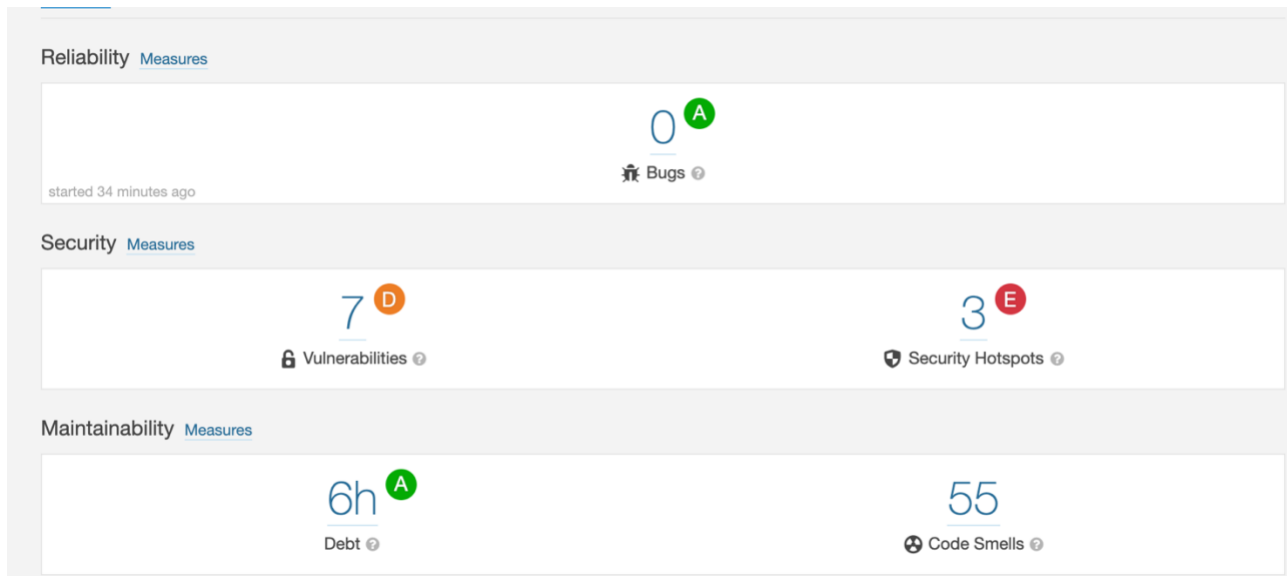


Figura 29

Configurazioni

Nel file `application.yml`, sono specificate tutte le direttive utili al sistema per interfacciarsi con i vari DB (H2, MySQL).

```
spring:
  profiles: test2
  jpa:
    properties:
      hibernate:
        jdbc:
          time_zone: UTC
        dialect: org.hibernate.dialect.MySQL5InnoDBDialect
      hibernate:
        ddl-auto: create-drop
  datasource:
    url: jdbc:mysql://localhost:3306/hibernate1?createDatabaseIfNotExist=true&useSSL=false&useLegacyDatetimeCode=false&serverTimezone=UTC
    username: operations
    password: operations
```

Figura 30: `Application.yml`, profilo `test2`

Nel file si osservano altri campi necessari per l'utilizzo di MySQL come:

1. `time_zone`
2. `dialect`: specifica a Hibernate il tipo di dialetto per comunicare con il relativo DB (MySQL in questo caso)
3. `hibernate.ddl-auto`: specifica la modalità con cui verrà gestito il database-schema in questione. Nel caso specifico `create-drop`, specifica che se non vi è un database (in questo caso `hibernate1`) lo crea e una volta terminata l'esecuzione del programma lo distrugge.
4. `url`: specifica molte informazioni tra cui il fatto di non usare il protocollo SSL e di creare il database se non è presente.
5. credenziali di accesso

Di seguito si osserva il più codice di configurazione per quanto riguarda il DB H2:

```
datasource:
  url: jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
  username: sa
  password: sa
  driver-class-name: org.h2.Driver
h2:
  console:
    enabled: true
    path: /h2-console
```

Figura 31: Application.yml, configurazione h2

Vi è ora la possibilità di eseguire dei test e osservare l'esito andando alla pagina web <http://localhost:8080/<context-path>/h2-console>.

Test

Per quanto riguarda la parte inerente i test case, sono stati creati dei test per l'analisi del modello di dominio e per la user interface.

Per quanto riguarda il modello di dominio è stato utilizzato il servizio di "org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest" che permette di testare in parziale isolamento il sistema di JPA.

```
@RunWith(SpringRunner.class)
@ComponentScan(basePackages = {"operations"})
@EnableAutoConfiguration(exclude = LiquibaseAutoConfiguration.class)
@DataJpaTest
```

Figura 32

Grazie alla configurazione del Bean h2Server è possibile utilizzare il DB h2 creando un canale tcp tra l'applicazione e il DB engine.

```
@Configuration
static class ContextConfiguration {
    @Bean(initMethod = "start", destroyMethod = "stop")
    public Server h2Server() throws SQLException {
        return Server.createTcpServer("-tcp", "-tcpAllowOthers", "-tcpPort", "9092");
    }
}
```

Figura 33

Vengono poi creati in una fase di setup degli oggetti che saranno poi salvati nel database e successivamente sarà testato l'effettiva presenza sul DB.


```

List<User> users = userService.findAll();
System.err.println("UserID    " + "UserOperation    " + "UserName");
assertEquals(users.get(0).getName(), actual: "name3");
assertEquals(users.get(1).getName(), actual: "name4");
assertEquals(users.get(2).getName(), actual: "name5");

```

Figura 34

Per quanto riguarda la User-Interface sono stati utilizzati i framework di Test: Selenium e Selenide.

Quest'ultimo rappresenta un Wrapper di Selenium.

Inoltre, oltre alle classiche dipendenze di Selenium e Selenide è stata aggiunta la dipendenza che si osserva nell'immagine sotto, necessaria per l'utilizzo dei due framework.

```

<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>30.1.1-jre</version>
</dependency>

```

Figura 35

Di seguito si osserva un esempio di test che. Controlla l'effettiva presenza del bottone "register" e testa la sua corretta funzionalità.

```

@Test
public void SelenideLogIn() {

    open( relativeOrAbsoluteUrl: "http://localhost/");
    open( relativeOrAbsoluteUrl: "http://localhost/");
    $(By.id("registerId")).click();
    $(By.id("registerId")).isDisplayed();
    $(By.xpath("//*[text()=' " + " Log-in " + "']")).click();
    SelenideElement by1 = $(By.xpath("//*[contains(text(),'Register')]"));
    by1.click();
    $(By.xpath("//*[contains(text(),'Full name')]")).sendKeys( ...charSequences: "prova")
}

```

Figura 36

Di seguito si osservano i vari casi di test di unità, d'integrazione e e2e. Data la struttura identica tra le varie classi di dominio e nei Controller sono state testate un'entità e un controller.

```

@Test
void findAllTest() {
    when(this.userRepository.findAll()).
        thenReturn(asList(this.user1, this.user2));
    assertThat(this.userService.findAll()).
        contains(this.user1, this.user2);
    assertThat(this.userService.findAll()).doesNotContain(this.user3);
}

@Test
void findById() {
    when(this.userRepository.findById(1L)).
        thenReturn(Optional.of(this.user1));
    assertThat(this.userService.findById(1L))
        .isSameAs(this.user1);
}

```

Figura 37

Come si osserva nell'immagine sopra sono riportati alcuni dei test effettuati sulla userService. Vi è l'utilizzo delle annotazioni @Mock e @injectMock necessarie alla simulazione del test di unità.



Uncommitted changes

master 68 ahead Merge branch 'UserMockTesting'

UserMockTesting Commit per allineamento branch

Creazione e conclusione UserTest (unit testing)

no message

Figura 38

Per la realizzazione dei vari test sono stati creati dei branch per ogni nuova classe di Test con successivo “merge” al branch master una volta conclusa la classe con un valore di coverage del 100%.

```

@Service
public class UserService {

    private static final String ENTITY_NAME = "User";
    private UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User save(User user) {
        return this.userRepository.save(user);
    }

    public User findBySubject(String subject) {
        return this.userRepository.findBySubject(subject).orElseThrow(RuntimeException::new);
    }

    public List<User> findAll() {
        return StreamSupport.stream(this.userRepository.findAll().spliterator(), false)
            .collect(Collectors.toList());
    }

    public User findById(Long id) {
        return this.userRepository.findById(id).orElseThrow(RuntimeException::new);
    }
}

```

Figura 39

Di seguito viene mostrato un esempio di Integration Test il Domain layer:

```
@RunWith(SpringRunner.class)
@ComponentScan(basePackages = {"operations"})
@EnableAutoConfiguration(exclude = LiquibaseAutoConfiguration.class)
@DataJpaTest
public class DomainDataJpaTest {
```

Figura 40

```
@Test
public void testQuery() throws InterruptedException {
    List<User> users = userService.findAll();
    System.err.println("UserID " + "UserOperation " + "UserName");
    assertEquals(users.get(0).getName(), actual: "name3");
    assertEquals(users.get(1).getName(), actual: "name4");
    assertEquals(users.get(2).getName(), actual: "name5");
}
```

Figura 41

Su questi test è stato poi eseguito il Plug-in di Pit Mutation testing necessario per trovare i mutanti vivi o uccisi. Per fare ciò è stato eseguito il comando `mvn org.pitest:pitest-maven:mutationCoverage`.

```
<dependency>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <version>1.7.0</version>
</dependency>

</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.pitest</groupId>
      <artifactId>pitest-maven</artifactId>
      <version>1.7.0</version>
    </plugin>
  </plugins>
</build>
```

Figura 42

Come ultima parte della revisione del progetto è stato utilizzato il tool di SonarQube per verificare la qualità del codice. In particolare è stato utilizzato il container su più moduli messo a disposizione online usufruibile attraverso il file `docker-compose.yml`. Sono di seguito riportate le prime righe del file:

```

version: "2"
services:
  sonarqube:
    image: sonarqube:8.2-community
    depends_on:
      - db
    ports:
      - "9000:9000"
    networks:
      - sonarnet
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://db:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar

```

Figura 43

Una volta lanciato il container attraverso il comando “Docker-compose up” e dopo aver fatto la build del progetto è possibile eseguire il comando “mvn sonar:sonar”, così da realizzare l’analisi statica del codice.

```

<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.sonarsource.scanner.maven</groupId>
        <artifactId>sonar-maven-plugin</artifactId>
        <version>3.7.0.1746</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

```

Figura 44

L’output prodotto riguarda è visibile nell’immagine in basso.

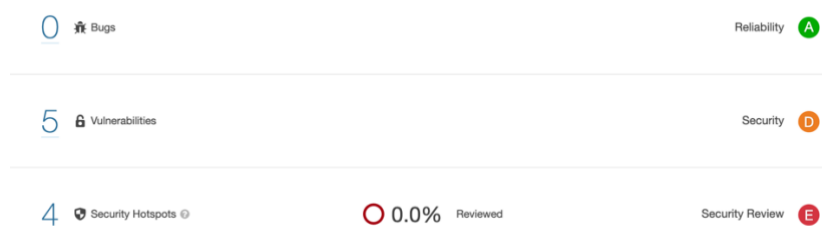


Figura 45

In particolare, sono presenti 5 vulnerabilità che riguardano la necessità di sostituire nei metodi del controller le entità con oggetti POJO o DTO:

Replace this persistent entity with a simple POJO or DTO object. Why is this an issue?	7 months ago • L36
Vulnerability Critical Open Not assigned 10min effort	cwe, owasp-a5, spring
Replace this persistent entity with a simple POJO or DTO object. Why is this an issue?	7 months ago • L66
Vulnerability Critical Open Not assigned 10min effort	cwe, owasp-a5, spring
business-logic-layer/_/backend/user/controllers/UserController.java	
Replace this persistent entity with a simple POJO or DTO object. Why is this an issue?	11 days ago • L55
Vulnerability Critical Open Not assigned 10min effort	cwe, owasp-a5, spring
Replace this persistent entity with a simple POJO or DTO object. Why is this an issue?	11 days ago • L69
Vulnerability Critical Open Not assigned 10min effort	cwe, owasp-a5, spring
Replace this persistent entity with a simple POJO or DTO object. Why is this an issue?	4 days ago • L108
Vulnerability Critical Open Not assigned 10min effort	cwe, owasp-a5, spring

Figura 46

Front-end docker

Dato il largo utilizzo di docker è stato pensato di creare una docker image anche per l'app di Angular relativa al Front-end. Ciò permette un'installazione più semplice così da poter testare l'app. Di seguito è riportato il codice del file di Docker.

```
FROM node:alpine as build-step
RUN mkdir -p /app
WORKDIR /app
COPY package.json /app
#importante che tutte le dipendenze del sistema siano specificate nel package.json e non ve ne siano installate manualmente
#altrimenti nella fase di dockeraggio non verranno ritrovate le rispettive dipendenze
RUN npm install -g @angular/cli@latest
RUN npm install --force
#importante mettere --force per i conflitti delle dipendenze
COPY . /app

RUN npm run build
# Stage 2
FROM nginx:1.17.1-alpine
COPY --from=build-step /app/dist/advanced-con-front-end /usr/share/nginx/html
#seguire poi le istruzioni a https://dzone.com/articles/how-to-dockerize-angular-app
```

Figura 47

In particolare, si osserva il sistema da su cui si appoggia l'app (node) che offre tutti i servizi necessari al funzionamento di questa.

Vi sono poi dei comandi quali "RUN npm install" e "RUN npm build" necessari a installare le dipendenze e eseguire la build del progetto.

Nell'immagine di seguito è riportato anche il contenuto del file .dockerignore. Tale file risulta molto importante in quanto la dimensione delle dipendenze è molto grande.

```
.git
.firebase
package-lock.json
.editorconfig
/node_modules
/e2e
/docs
.gitignore
*.zip
*.md
```

Figura 48

RXJS

Nella parte finale della relazione, vi è un accenno sul funzionamento delle richieste e risposte fatte nel lato front-end.

```
public postUserWithHeader(httpOptions: any): Observable<any> {  
    return this.http.post<any>(url: this.url + '/operations-backend/api/user', httpOptions);  
}
```

Figura 49: PostUser con Observable

Nell'immagine 45 si osserva un esempio di post-request. Si osserva una dinamica simile a quella descritta precedentemente con l'aggiunta di un secondo argomento al metodo post.

L'oggetto httpOptions passato alla funzione, infatti, permette di specificare l'header della richiesta oltre che altri parametri utili:

```
public httpOptions = {  
    headers: new HttpHeaders({ headers: {  
  
        'x-auth-subject': 'soggettoProva',  
        'x-auth-roles': 'ruoloProva ',  
        'x-auth-email': 'example@example',  
        'x-auth-username': 'userexample',  
        'x-auth-userid': 'userexample'  
    })  
};
```

Figura 50: Header del pacchetto HTTP

Sitografia

- [1] https://en.wikipedia.org/wiki/Non-functional_requirement
- [2] <https://itzone.com.vn/en/article/spring-data-jpa-1-overview-and-concepts/>
- [3] <https://pratikchanchpara.medium.com/triggering-a-jenkins-build-automatically-when-a-push-to-bitbucket-introduced-b90dec12f893>
- <https://stackoverflow.com/questions/49614262/bit-bucket-jenkins-webhooks-invalid-url>
- [5] <https://dev.mysql.com/doc/refman/8.0/en/grant.html#grant-overview>
- [6] http://www.h2database.com/html/features.html#in_memory_databases
- [9] <https://dariopironi.com/angular-httpclient-la-guida-pratica/>
- <https://rxjs-dev.firebaseapp.com/guide/operators>
- [10] [https://it.wikipedia.org/wiki/Analisi_\(ingegneria_del_software\)](https://it.wikipedia.org/wiki/Analisi_(ingegneria_del_software))
- [11] <https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api>
- [12] <https://dzone.com/articles/dependency-injection-in-spring>
- [13] <https://dzone.com/articles/difference-between-beanfactory-and-applicationcont>
- [14] <https://medium.com/swlh/spring-vs-spring-boot-826fcfe21522>

