Università di Catania

PROGETTO
DISTRIBUITED SYSTEM AND BIG DATA

Sviluppo di una piattaforma per la gestione e prenotazione degli ordini di un ristorante

Dipartimento di Ingegneria Elettrica, Elettronica e Informatica
Università degli studi di Catania
Anno accademico 2021/2022

Studenti:
Pierluigi Gaglio (1000025301)
Luca Occhipinti (1000026003)

SOMMARIO

1. INTRODUZIONE

Il progetto sviluppato verte sulla realizzazione di una applicazione web per la gestione e prenotazione degli ordini di un ristorante. Per l'implementazione si è previsto l'utilizzo di Spiring MVC, Spring Data Mongo DB e Mongo DB.

Tra le **configurazioni** proposte si è scelta la seguente:

Docker	**Kubernetes**	**Saga Coreography**	**API Gateway**	**Whitebox monitoring (Prometheus)**

In particolare, il progetto è costituito dai **tre microservizi:**

- Gestione pietanze → *PietanzeMicroservice,*
- Gestione ordini → *OrdiniMicroservice,*
- Gestione menù → *MenuMicroservice,*

attraverso la quale è possibile gestire le prenotazioni degli ordini di un ristorante.

2. PIETANZE MICROSERVICE

Tale microservizio consente di gestire - mediante operazioni di inserimento, eliminazione e visualizzazione – le pietanze che comporranno il menù e gli ordini.

2.1 STRUTTURA DELLE CLASSI

2.1.1 MODEL

La classe utilizzata per la realizzazione del modello per la gestione del database è la seguente:

```
@Document(collection = "Pietanza")
public class Pietanza {

    @Id
    @JsonSerialize(using= ToStringSerializer.class)
    private ObjectId _id;
    private String nome;
    private double prezzo;
```

Figura 1: Modello Pietanze

La classe *Pietanza* è stata realizzata al fine di definire una struttura con le caratteristiche dei piatti che potranno essere ordinati.

2.1.2 CLASSI DI SUPPORTO

Tali classi di supporto sono:

- **MenuCreate:** classe realizzata al fine di fornire una struttura per i messaggi inviati sul **topic "menu",** i quali serviranno per verificare se le pietanze inserite nel menù esistono nel database associato al microservizio.

```
public class MenuCreate implements Serializable {

    private ObjectId menu_id;
    private List<ObjectId> pietanze;
    private String stato;
    private String nome;
```

Figura 2: classe supporto MenuCreate

- **OrderCreate:** classe realizzata al fine di fornire una struttura per i messaggi inviati sul **topic "order"**, i quali serviranno per verificare se le pietanze inserite nell'ordine esistono nel database associato al microservizio.

```
public class OrderCreate implements Serializable {

    private ObjectId order_id;
    private List<ObjectId> pietanze;
    private String stato;
```

Figura 3: classe supporto OrderCreate

2.2 SPRING BOOT

L'utilizzo di Spring Boot ha permesso l'implementazione di diversi componenti con la possibilità di interagire mediante il pattern Model-View-Controller.

2.2.1 CONTROLLER PIETANZA

Il *ControllerPietanza* è il componente che applica le funzioni del Controller previste dal pattern MVC con il compito di ricevere le richieste provenienti dal client e di reagire eseguendo operazioni sui dati e quindi sul database.

Sono state implementate le seguenti **API REST**:

- **POST:** permette di aggiungere una pietanza al database Mongo. Questa è raggiungibile attraverso il **path** "/pietanze/" alla **porta** "8083" con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/pietanze/pietanze/".

```
@PostMapping(path="/")
public @ResponseBody ResponseEntity<Object> addPietanza(@RequestBody Pietanza pietanza){
    visitCounter.increment();

    Pietanza pietanza1 = repository.findByNome(pietanza.getNome());

    if(pietanza1 == null){
        repository.save(pietanza);
        return new ResponseEntity<Object>("Pietanza inserita correttamente.", HttpStatus.OK);
    }else{
        return new ResponseEntity<Object>("Impossibile inserire due pietanze con lo stesso nome", HttpStatus.OK);
    }

}
```

Figura 4: POST Pietanze

- **GET:**

    - La prima funzione della figura seguente, permette di visualizzare tutte le pietanze contenute nel database Mongo. Questa è raggiungibile attraverso il **path** "/pietanze/" alla **porta** "8083" con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/pietanze/pietanze/".

    - La seconda visualizza una singola pietanza attraverso il suo ID. Questa è raggiungibile attraverso il **path** "/pietanze/id" alla **porta** "8083" con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/pietanze/pietanze/id".

```java
@GetMapping(path="/", produces = MediaType.APPLICATION_JSON_VALUE)
public @ResponseBody
ResponseEntity<Object> getAllPietanze(){
    visitCounter.increment();

    List<Pietanza> pietanza = repository.findAll();

    if (pietanza.size() != 0) {
        return new ResponseEntity<Object>(pietanza, HttpStatus.OK);
    }else{
        throw new PietanzaNotFoundException();
    }

}

@GetMapping(path="/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
public @ResponseBody
ResponseEntity<Object> getPietanzaById(@PathVariable ObjectId id){
    visitCounter.increment();

    Pietanza pietanza = repository.findPietanzaBy_id(id);

    if (pietanza != null) {
        return new ResponseEntity<Object>(pietanza, HttpStatus.OK);
    }else{
        throw new PietanzaNotFoundException();
    }

}
```

Figura 5: GET Pietanze

- **DELETE:** permette di rimuovere una pietanza dal database Mongo attraverso il suo ID. Questa è raggiungibile attraverso il **path** "/pietanze/id" alla **porta** "8083" con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/pietanze/pietanze/id".

```
@DeleteMapping(path="/{id}")
public @ResponseBody
ResponseEntity<Object> deletePietanza(@PathVariable ObjectId id){
    visitCounter.increment();

    Pietanza pietanza = repository.findPietanzaBy_id(id);

    if (pietanza != null) {
        repository.delete(pietanza);
        return new ResponseEntity<Object>("Pietanza cancellata correttamente.", HttpStatus.OK);
    }else{
        throw new PietanzaNotFoundException();
    }

}
```

Figura 6: DELETE Pietanze

2.3 MONGO

Per la realizzazione del database è stato utilizzato MongoDB, nonché un database non relazionale. Per raggiungere questo scopo, è stata creata un'interfaccia PietanzeRepository che estende la MongoRepository. All'interno dell'interfaccia sono stati inseriti i metodi utilizzati al fine di soddisfare le richieste.

```
public interface PietanzeRepository extends MongoRepository<Pietanza, ObjectId>{

    public Pietanza findByNome(String nome);

    public Pietanza findBy_id(ObjectId _id);

    public Pietanza findPietanzaBy_id(ObjectId _id);
}
```

Figura 7: PietanzeRepository

Inoltre, al fine di risolvere un'eccezione generata da Spring "Exception opening socket" è stato necessaria la creazione di un ApplicationConfiguration con lo scopo di risolvere l'eccezione e configurare manualmente la connessione mongo.

```java
@EnableMongoRepositories
@SpringBootApplication(
        exclude = {
                MongoAutoConfiguration.class,
                MongoDataAutoConfiguration.class,
                MongoReactiveDataAutoConfiguration.class,
                EmbeddedMongoAutoConfiguration.class
        }
)
@AutoConfigureAfter({EmbeddedMongoAutoConfiguration.class, MongoDataAutoConfiguration.class, MongoAutoConfiguration.class})
public class ApplicationConfiguration extends AbstractMongoClientConfiguration {

    @Value(value = "${MONGODB_HOSTNAME}")
    private String host;

    /*@Value(value="${MONGO_USER}")
    private String mongoUser;
    @Value(value="${MONGO_PASS}")
    private String mongoPass;
    @Value(value="${MONGO_AUTH_DB}")
    private String mongoAuthDb;*/

    @Value(value="${MONGODB_PORT}")
    private String port;

    @Value(value="${MONGO_DB_NAME}")
    private String dbname;

    public ApplicationConfiguration() {

    }

    @Override
    @Bean
    public MongoClient mongoClient() {

        //String s = String.format("mongodb://%s:%s@%s:%s/%s", user, pass, host, port, dbname);
        String s = String.format("mongodb://%s:%s/%s", host, port, dbname);
        return MongoClients.create(s);
    }

    @Override
    protected String getDatabaseName() {
        return dbname;
    }
}
```

Figura 8: ApplicationConfiguration

In particolare, abbiamo deciso di non includere i parametri di autenticazione in quanto mongo non supporta l'inserimento automatico dell'utente all'interno della tabella admin. Per cui, a fini di sviluppo, si è deciso di ometterli.

2.4 GESTIONE DELLE ECCEZIONI

Nel caso in cui una pietanza non esiste, è necessario generare un errore con status code 404 corrispondente a ***NOT_FOUND***. E' stata implementata l'eccezione personalizzata *PietanzaNotFoundException* nella quale mediante l'annotation *@ResponseStatus* è stato impostato lo status code.

```
@ResponseStatus(code = HttpStatus.NOT_FOUND)
public class PietanzaNotFoundException extends RuntimeException{
}
```

Figura 9: Eccezione Pietanze Not_Found

2.5 DOCKER

Per la gestione dei microservizi è stato utilizzato Docker, il quale permette il loro sviluppo all'interno dei container.

2.5.1 DOCKEFILE

Il DockerFile per il microservizio *pietanza* è così formato:

```
FROM maven:3-jdk-8 AS builder
WORKDIR /project
COPY . .
RUN mvn package


FROM openjdk:8-jdk-alpine

WORKDIR /app
#viene copiato l'artefatto costruito dal builder nell'immagine docker
COPY --from=builder /project/target/PietanzeMicroservice-0.0.1-SNAPSHOT.jar ./PietanzeMicroservice.jar
ENTRYPOINT ["/bin/sh", "-c"]
CMD [ "java -jar PietanzeMicroservice.jar" ]
```

Figura 10: DockerFile Pietanze

In questo caso viene utilizzata un'implementazione multi-stage. Il file contiene una serie di comandi per la creazione dell'immagine a partire da un contesto di applicazione, ovvero una directory locale. Questo permetterà di costruire un'immagine di base, utilizzando il comando FROM. Il comando RUN serve ad eseguire in fase di building dell'immagine un comando, in questo caso il *mvn package*. Si prosegue con l'esecuzione del container mediante il CMD.

2.5.2 DOCKER-COMPOSE

Il *docker-compose* viene inserito all'interno della cartella complessiva del progetto quindi risulterà essere comune a tutti e tre i microservizi.

Vengono creati i seguenti container:

```yaml
services:
  pietanze-service-db:
    image: mongo
    restart: always
    #environment:
     #MONGO_INITDB_ROOT_USERNAME: root
     #MONGO_INITDB_ROOT_PASSWORD: toor
    ports:
      - 27017:27017
    volumes:
      - mongo-db-data-pietanze:/data/db

  pietanzemicroservice:
    build:
      context: .
      dockerfile: PietanzeMicroservice/Dockerfile
    ports:
      - 8083:8080
    restart: always
    environment:
      <<:  *mongo-credentials-pietanze
      KAFKA_GROUP_ID: group-consumer-pietanze
      KAFKA_ADDRESS: kafka:9092
      KAFKA_TOPIC_ORDERS: "order"
      KAFKA_TOPIC_MENU: "menu"
```

Figura 11: docker-compose Pietanze

2.6 KUBERNETES

Kubernetes è una piattaforma portatile, estensibile e open-source per la gestione di carichi di lavoro e servizi containerizzati, in grado di facilitare sia la configurazione dichiarativa che l'automazione. La sua configurazione ha permesso di automatizzare il processo di creazione e funzionamento dei microservizi.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pietanze-service-db
spec:
  selector:
    matchLabels:
      app: pietanze-service-db
  template:
    metadata:
      labels:
        app: pietanze-service-db
    spec:
      containers:
        - name: pietanze-service-db
          image: mongo:latest
          ports:
            - containerPort: 27017
          volumeMounts:
            - mountPath: /data/db
              name: mongo-db-data-pietanze
      volumes:
        - name: mongo-db-data-pietanze
          persistentVolumeClaim:
            claimName: mongo-db-data-pietanze
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pietanze-microservice
spec:
  selector:
    matchLabels:
      app: pietanze-microservice
  template:
    metadata:
      labels:
        app: pietanze-microservice
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "8080"
        prometheus.io/path: "/actuator/prometheus"
    spec:
      containers:
      - name: pietanze-microservice
        image: pierluca97/pietanzerepo:latest
        ports:
          - containerPort: 8080
        envFrom:
          - configMapRef:
              name: pietanze-service-env-file
```

Figura 12: Deployment database e microservizio Pietanze

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pietanze-microservice
spec:
  ports:
    - protocol: TCP
      name: http-traffic
      port: 8080
      targetPort: 8080
  selector:
    app: pietanze-microservice
```

Figura 15: Service microservizio Pietanze

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pietanze-service-db
spec:
  ports:
    - port: 27017
  selector:
    app: pietanze-service-db
```

Figura 14: Service database Pietanze

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: pietanze-service-env-file
data:
  KAFKA_GROUP_ID: group-consumer-pietanze
  KAFKA_ADDRESS: kafkabroker:9092
  KAFKA_TOPIC_ORDERS: "order"
  KAFKA_TOPIC_MENU: "menu"
  MONGODB_HOSTNAME: pietanze-service-db
  MONGODB_PORT: "27017"
  MONGO_DB_NAME: mydbpietanze
```

Figura 13: Variabili d'ambiente

2.6.1 API GATEWAY

Il pattern API Gateway viene implementato sfruttando il componente di Kubernetes chiamato **Ingress.** Questo permetterà di inoltrare la richiesta effettuata al microservizio corretto.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: spring-ingress
  annotations:
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-methods: "PUT, GET, POST, OPTIONS, DELETE"
    nginx.ingress.kubernetes.io/cors-allow-origin: "*"
    nginx.ingress.kubernetes.io/cors-allow-credentials: "true"
    nginx.ingress.kubernetes.io/cors-allow-headers: "Content-Type"
    nginx.ingress.kubernetes.io/rewrite-target: "/$2"
spec:
  rules:
    - host: cluster.dsbd.it
      http:
        paths:
          - path: /menu(/|$)(.*)
            pathType: ImplementationSpecific
            backend:
              service:
                name: menu-microservice
                port:
                  number: 8080
          - path: /ordini(/|$)(.*)
            pathType: ImplementationSpecific
            backend:
              service:
                name: ordini-microservice
                port:
                  number: 8080
          - path: /pietanze(/|$)(.*)
            pathType: ImplementationSpecific
            backend:
              service:
                name: pietanze-microservice
                port:
                  number: 8080
```

Figura 16: Ingress

2.6.2 PROMETHEUS E GRAFANA

Per l'archiviazione di metriche e Time series viene utilizzato il tool di monitoraggio e alerting Prometheus. L'utilizzo di Grafana permetterà di accedere in maniera immediata ai dati immagazzinati da Prometheus. Effettuando il collegamento trai i due, è possibile costruire delle *query* parametrizzabili e visualizzare i grafici che mettono in relazione le metriche. In particolare, la metrica analizzata è la **visit_counter** la quale conteggia il numero di richieste effettuate ad un singolo microservizio.Il tutto è stato configurato creando il **namespace monitoring** all'interno del quale, mediante dei file yaml, vengono attivati sia Prometheus che Grafana.Facendo riferimento al paragrafo $2.6 → Figura 12, è possibile notare la presenza delle annotation che permettono a Prometheus di estrapolare i dati relativi al microservizio. Questo sarà visibile all'end-point "cluster.dsbd.it:30000". Accedendo all'end-point "cluster.dsbd.it:32000", sarà possibile effettuare la query al fine di prelevare i dati da analizzare. Questi ultimi saranno analizzati mediante l'algoritmo **Holt-Winters** al fine di predire come evolverà la serie in futuro.

I risultati ottenuti dopo un monitoraggio di 30 minuti sono i seguenti:

E' stato trovato che la seasonability della serie è 12, i residui sono approssimativamente a media nulla, il trend risulta essere lineare e le predizioni sono abbastanza accurate seppur su un numero ristretto di campioni.

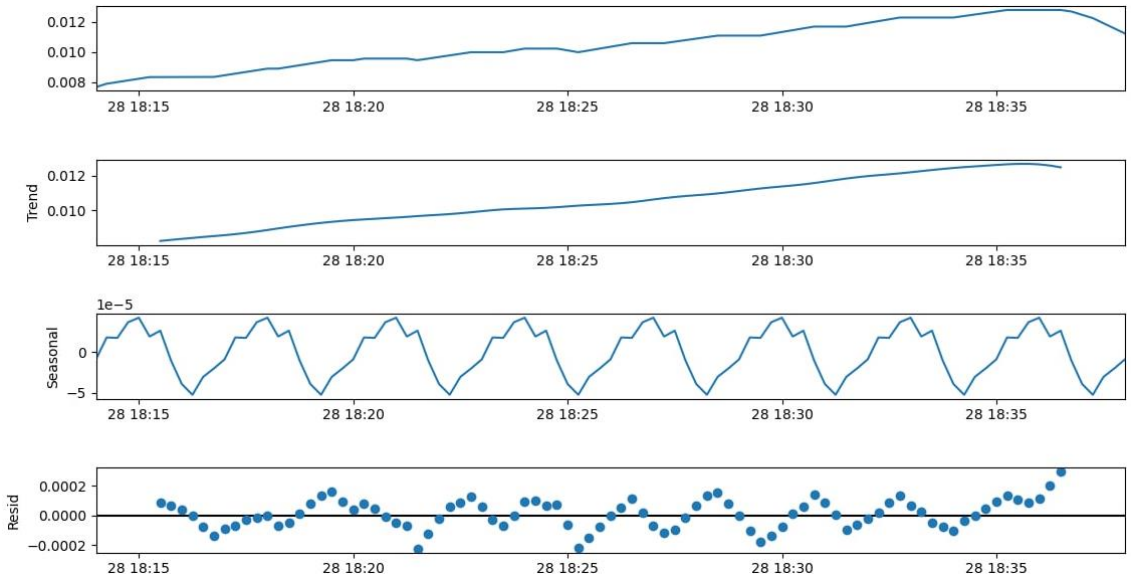

Figura 17: Trend, Stagionalità, Residui Pietanze

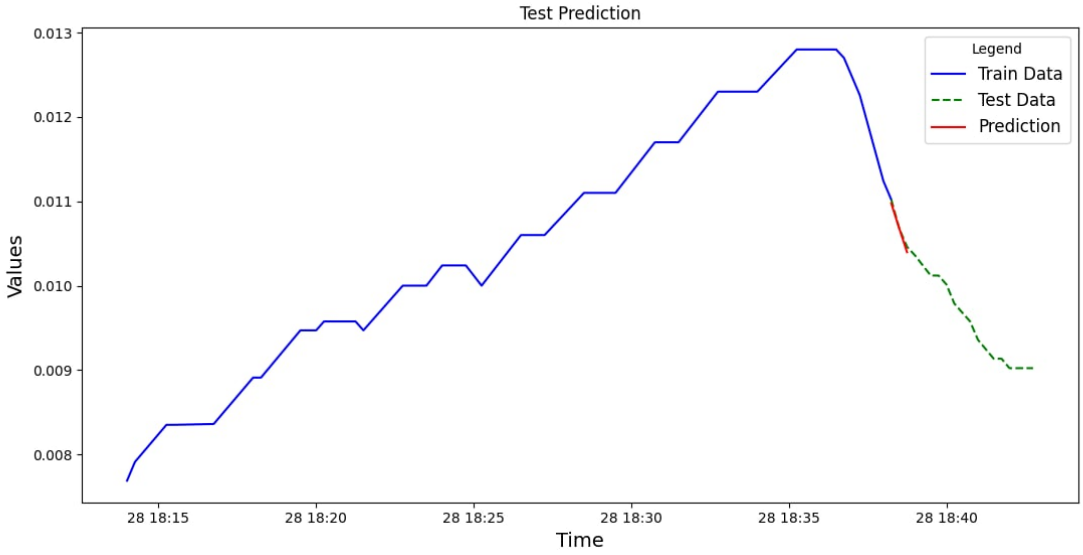

Figura 18: Test prediction Pietanze

I risultati sui test data risultano abbastanza precisi e dunque facendo una predizione futura il risultato ottenuto è il seguente:

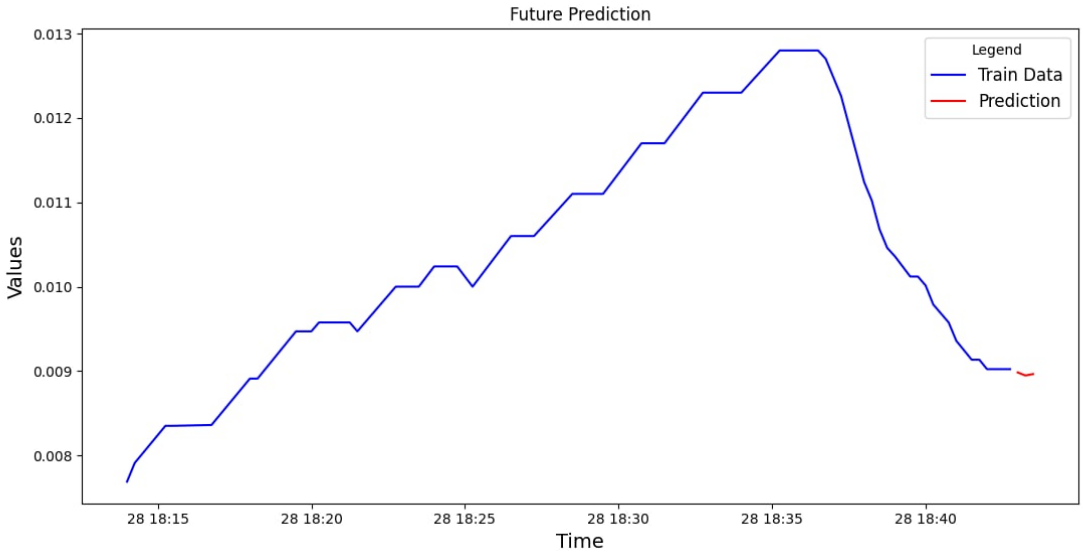

Figura 19: Future prediction Pietanze

3. ORDINI MICROSERVICE

Tale microservizio consente di gestire - mediante operazioni di inserimento, eliminazione e visualizzazione – gli ordini.

3.1 STRUTTURA DELLE CLASSI

3.1.1 MODEL

La classe utilizzata per la realizzazione del modello per la gestione del database è la seguente:

```
@Document(collection = "Ordine")
public class Ordine {
    @Id
    @JsonSerialize(using= ToStringSerializer.class)
    private ObjectId _id;
    private String stato;
    private List<ObjectId> pietanze;
```

Figura 20: Modello Ordini

La classe *Ordini* è stata realizzata al fine di definire una struttura costituita dalle pietanze che sono state ordinate.

3.1.2 CLASSI DI SUPPORTO

Tale classe di supporto è:

- **OrderCreate:** classe realizzata al fine di fornire una struttura per i messaggi inviati sul **topic "order"**, i quali serviranno per verificare se le pietanze inserite nell'ordine esistono nel database associato al microservizio.

```
public class OrderCreate implements Serializable {

    private ObjectId order_id;
    private List<ObjectId> pietanze;
    private String stato;
```

Figura 21: classe supporto OrderCreate

3.2 SPRING BOOT

3.2.1 CONTROLLER ORDINE

Il *ControllerOrdine* è il componente che applica le funzioni del Controller previste dal pattern MVC con il compito di ricevere le richieste provenienti dal client e di reagire eseguendo operazioni sui dati e quindi sul database.

Sono state implementate le seguenti **API REST**:

- **POST:** permette di aggiungere un ordine al database Mongo. Questo è raggiungibile attraverso il **path** "/ordine" alla **porta "**8082**"** con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/ordini/ordine".

```
@PostMapping(path="/ordine")
public @ResponseBody String addOrdine(@RequestBody Ordine ordine) throws InterruptedException {

    visitCounter.increment();

    Ordine new_ordine = repository.save(ordine);
    OrderCreate order = new OrderCreate(new_ordine.get_id(), ordine.getPietanze(), "order_create");
    kafkaTemplate.send("order", "order_create", new Gson().toJson(order));

    System.out.println("ORDINE: " + new_ordine.get_id());

    Thread.sleep(10000);

    Ordine order_finale = repository.findOrdineBy_id(new_ordine.get_id());

    if(order_finale != null && order_finale.getStato().equals("order_confirmed")){

        order_finale.setStato("order_confirmed");

        repository.save(order_finale);

        esito = "Ordine creato correttamente";

    }else{

        repository.deleteById(new_ordine.get_id());

        esito = "Ordine fallito";

    }

    return esito;
}
```

Figura 22: POST Ordini

- **GET:**
  - La prima funzione della figura seguente, visualizza un singolo ordine attraverso il suo ID. Questo è raggiungibile attraverso il **path** "/ordine/id" alla **porta "**8082**"** con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/ordini/ordine/id".
  - La seconda, permette di visualizzare tutti gli ordini contenuti nel database Mongo. Questo è raggiungibile attraverso il **path** "/ordine" alla **porta "**8082**"** con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/ordini/ordine".

```
@GetMapping(path="/ordine/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
public @ResponseBody
ResponseEntity<Object> getOrdineByID(@PathVariable("id") ObjectId id){

    visitCounter.increment();

    Ordine ordine = repository.findBy_id(id);

    if (ordine != null) {
        return new ResponseEntity<Object>(ordine, HttpStatus.OK);
    }else{
        throw new OrderNotFoundException();
    }

}

@GetMapping(path="/ordine", produces = MediaType.APPLICATION_JSON_VALUE)
public @ResponseBody
ResponseEntity<Object> getAllOrdini(){

    visitCounter.increment();

    List<Ordine> ordine = repository.findAll();

    if (ordine.size() != 0) {
        return new ResponseEntity<Object>(ordine, HttpStatus.OK);
    }else{
        throw new OrderNotFoundException();
    }

}
```

Figura 23: GET Ordini

- **DELETE:** permette di rimuovere un ordine dal database Mongo attraverso il suo ID. Questo è raggiungibile attraverso il **path** "/ordine/id" alla **porta "**8082**"** con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/ordini/ordine/id".

```
@DeleteMapping(path="/ordine/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
public @ResponseBody
ResponseEntity<Object> deleteOrdine(@PathVariable("id") ObjectId id){

    visitCounter.increment();

    Ordine ordine = repository.findBy_id(id);

    if (ordine != null) {
        repository.deleteById(id);

        return new ResponseEntity<Object>("Cancellazione dell'ordine effettuata correttamente.", HttpStatus.OK);
    }else{
        throw new OrderNotFoundException();
    }

}
```

Figura 24: DELETE Ordine

3.2.2 SAGA COREOGRAPHY

Per l'implementazione del pattern Saga Coreography, si è deciso di utilizzare come meccanismo di invio dei messaggi tra i due microservizi: *KAFKA*. Questo è stato opportunatamente configurato mediante dei file di configurazione rispettivamente per il **produttore** e per il **consumatore**, definendo inoltre l'opportuno topic chiamato **ORDER**.

Nella figura 22 → $3.2.1 relativa alla POST, è possibile notare come nel momento in cui viene salvato l'ordine all'interno del database, questo venga successivamente inviato come messaggio sul topic dopo essere stato serializzato. Verrà consumato opportunatamente dal microservizio pietanze, che si occuperà di verificare che le pietanze inserite esistano realmente all'interno del database.

```java
@KafkaListener(topics = {"${KAFKA_TOPIC_ORDERS}","${KAFKA_TOPIC_MENU}"}, groupId = "${KAFKA_GROUP_ID}")
public String listenValidation(ConsumerRecord<String, String> record) {
    if(record != null){
        System.out.println(record.key());
        if(record.key().equals("order_create")) {
            System.out.println("Messaggio ricevuto sul topic order con chiave order_create");
            OrderCreate order_create = new Gson().fromJson(record.value(), OrderCreate.class);

            System.out.println(order_create.toString());

            List<ObjectId> pietanze = order_create.getPietanze();

            for(int j = 0; j < pietanze.size(); j++) {
                System.out.println(pietanze.get(j));
            }

            System.out.println("PRIMA DEL FOR");

            for(int i = 0; i < pietanze.size(); i++) {

                System.out.println("DENTRO IL FOR");

                Pietanza p = pietanzeRepository.findPietanzaBy_id(pietanze.get(i));
                //System.out.println(p.toString());

                if(p == null){

                    System.out.println("SONO DENTRO L IF");

                    order_create.setStato("order_failed");
                    kafkaTemplate.send("order", "order_failed", new Gson().toJson(order_create));
                    return "Ordine fallito.";
                }
            }
            order_create.setStato("order_confirmed");
            kafkaTemplate.send("order", "order_confirmed", new Gson().toJson(order_create));
            return "Ordine confermato";
```

Figura 25: Consumer Kafka Pietanze

In caso di esito positivo, verrà settato lo stato dell'ordine in confirmed, altrimenti in failed e successivamente verrà inviato nuovamente un messaggio sul topic. Quest'ultimo verrà consumato dal microservizio Ordine stesso, il quale verificando lo stato dell'ordine, lo aggionerà all'interno del database.

Per abilitare il meccanismo di rollback, è stata inserite una sleep di pochi millisecondi nel controllore al fine di attendere l'opportuno settaggio dello stato. Nel caso in cui ciò avvenga con stato "**order_confirmed**", allora verrà salvato all'interno del database. Altrimenti se lo stato è pari a "**order_failed**" o

non è stato ancora aggiornato, verrà eliminato, al fine di mantenere una coerenza dei dati.

```
@KafkaListener(topics = "${KAFKA_TOPIC_ORDERS}", groupId = "${KAFKA_GROUP_ID}")
public void listenOrderValidation(ConsumerRecord<String, String> record) {
    if(record != null){

        if(record.key().equals("order_confirmed")) {

            OrderCreate ordine = new Gson().fromJson(record.value(), OrderCreate.class);

            Ordine order_confirmed = new Ordine();

            order_confirmed.set_id(ordine.getOrder_id());
            order_confirmed.setPietanze(ordine.getPietanze());
            order_confirmed.setStato("order_confirmed");

            System.out.println("KAFKA LISTENER ORDINI: " + ordine.toString());

            Ordine prova = repository.save(order_confirmed);

            System.out.println("KAFKA LISTENER ORDINI DOPO SAVE: " + prova.toString());

        }else{

            OrderCreate ordine = new Gson().fromJson(record.value(), OrderCreate.class);

            Ordine order_failed = new Ordine();

            order_failed.set_id(ordine.getOrder_id());
            order_failed.setPietanze(ordine.getPietanze());
            order_failed.setStato("order_failed");

            System.out.println("KAFKA LISTENER ORDINI: " + ordine.toString());

            Ordine prova = repository.save(order_failed);

            System.out.println("KAFKA LISTENER ORDINI DOPO SAVE: " + prova.toString());
```

Figura 26: Consumer Kafka Ordini

3.3 MONGO

Per la realizzazione del database è stato utilizzato MongoDB, nonché un database non relazionale. Per raggiungere questo scopo, è stata creata un'interfaccia OrdineRepository che estende la MongoRepository. All'interno dell'interfaccia sono stati inseriti i metodi utilizzati al fine di soddisfare le richieste.

```java
public interface OrdineRepository extends MongoRepository<Ordine, ObjectId>{

    public Ordine findBy_id(ObjectId _id);

    public Ordine findOrdineBy_id(ObjectId _id);

}
```

Figura 27: OrdineRepository

Inoltre, al fine di risolvere un'eccezione generata da Spring "Exception opening socket" è stato necessaria la creazione di un ApplicationConfiguration. Si faccia riferimento al paragrafo $2.3.

3.4 GESTIONE DELLE ECCEZIONI

Nel caso in cui un ordine non esiste, è necessario generare un errore con status code 404 corrispondente a ***NOT_FOUND***. E' stata implementata l'eccezione personalizzata *OrderNotFoundException* nella quale mediante l'annotation *@ResponseStatus* è stato impostato lo status code.

```java
@ResponseStatus(code = HttpStatus.NOT_FOUND)
public class OrderNotFoundException extends RuntimeException{
}
```

Figura 28: Eccezione Ordini Not_Found

3.5 DOCKER

3.5.1 DOCKERFILE

Il DockerFile per il microservizio *ordini* è così formato:

```
FROM maven:3-jdk-8 AS builder
WORKDIR /project
COPY . .
RUN mvn package


FROM openjdk:8-jdk-alpine

WORKDIR /app
#viene copiato l'artefatto costruito dal builder nell'immagine docker
COPY --from=builder /project/target/OrdiniMicroservice-0.0.1-SNAPSHOT.jar ./OrdiniMicroservice.jar
ENTRYPOINT ["/bin/sh", "-c"]
```

Figura 29: DockerFile Ordini

Per la sua definizione fare riferimento al paragrafo $2.5.1.

3.5.2 DOCKER-COMPOSE

Il *docker-compose* viene inserito all'interno della cartella complessiva del progetto, quindi risulterà essere comune a tutti e tre i microservizi.

Vengono creati i seguenti container:

```yaml
ordini-service-db:
  image: mongo
  command: mongod --port 27019
  restart: always
  #environment:
    #MONGO_INITDB_ROOT_USERNAME: root
    #MONGO_INITDB_ROOT_PASSWORD: toor
  ports:
    - 27019:27019
  volumes:
    - mongo-db-data-ordini:/data/db

ordinimicroservice:
  build:
    context: .
    dockerfile: OrdiniMicroservice/Dockerfile
  ports:
    - 8082:8080
  restart: always
  environment:
    <<:  *mongo-credentials-ordini
    KAFKA_GROUP_ID: group-consumer-ordini
    KAFKA_ADDRESS: kafka:9092
    KAFKA_TOPIC_ORDERS: "order"
```

Figura 30: docker-compose Ordini

3.6 KUBERNETES

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ordini-service-db
spec:
  selector:
    matchLabels:
      app: ordini-service-db
  template:
    metadata:
      labels:
        app: ordini-service-db
    spec:
      containers:
      - name: ordini-service-db
        image: mongo:latest
        ports:
          - containerPort: 27017
        volumeMounts:
          - mountPath: /data/db
            name: mongo-db-data-ordini
      volumes:
      - name: mongo-db-data-ordini
        persistentVolumeClaim:
          claimName: mongo-db-data-ordini
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ordini-microservice
spec:
  selector:
    matchLabels:
      app: ordini-microservice
  template:
    metadata:
      labels:
        app: ordini-microservice
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "8080"
        prometheus.io/path: "/actuator/prometheus"
    spec:
      containers:
      - name: ordini-microservice
        image: pierluca97/ordinirepo:latest
        ports:
        - containerPort: 8080
        envFrom:
          - configMapRef:
              name: orders-service-env-file
```

Figura 31: Deployment database e microservice

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: orders-service-env-file
data:
  KAFKA_GROUP_ID: group-consumer-ordini
  KAFKA_ADDRESS: kafkabroker:9092
  KAFKA_TOPIC_ORDERS: "order"
  MONGODB_HOSTNAME: ordini-service-db
  MONGODB_PORT: "27017"
  MONGO_DB_NAME: mydbordini
```

Figura 34: Variabili d'ambiente

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ordini-microservice
spec:
  ports:
    - protocol: TCP
      name: http-traffic
      port: 8080
      targetPort: 8080
  selector:
    app: ordini-microservice
```

Figura 32: Service microservizio

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ordini-service-db
spec:
  ports:
    - port: 27017
  selector:
    app: ordini-service-db
```

Figura 33: Service database

3.6.1 API GATEWAY

Il pattern API Gateway viene implementato sfruttando il componente di Kubernetes chiamato **Ingress.** Per la sua definizione fare riferiento al paragrafo $2.6.1.

3.6.2 PROMETHEUS E GRAFANA

I risultati ottenuti dopo un monitoraggio di 30 minuti sono i seguenti:

E' stato trovato che la seasonability della serie è 15, i residui sono approssimativamente a media nulla e le predizioni sono abbastanza accurate seppur su un numero ristretto di campioni.

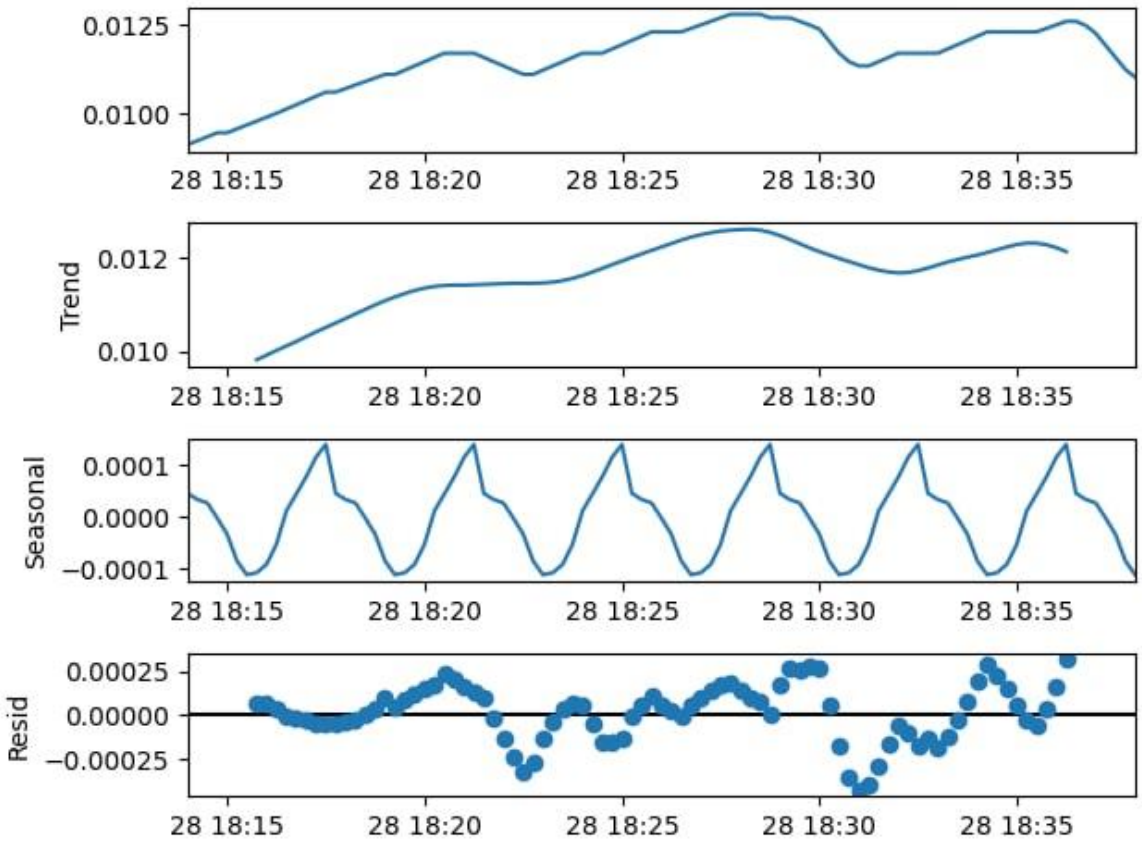

Figura 36: Trend, Stagionalità, Residui Ordini

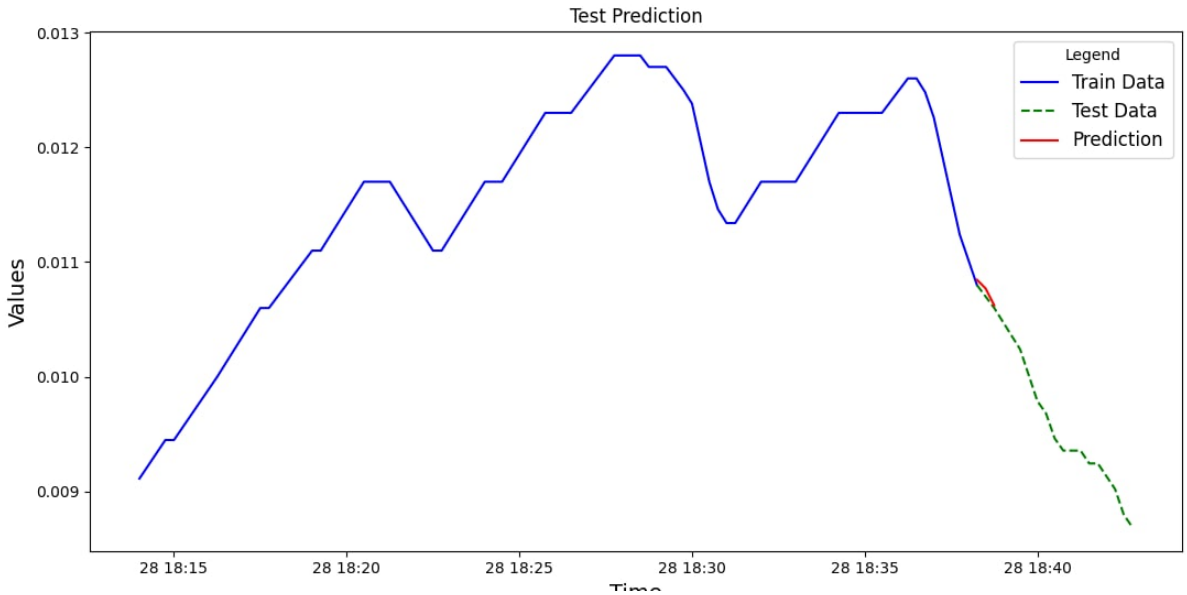

Figura 37: Test Prediction Ordini

I risultati sui test data risultano abbastanza precisi e dunque facendo una predizione futura il risultato ottenuto è il seguente:

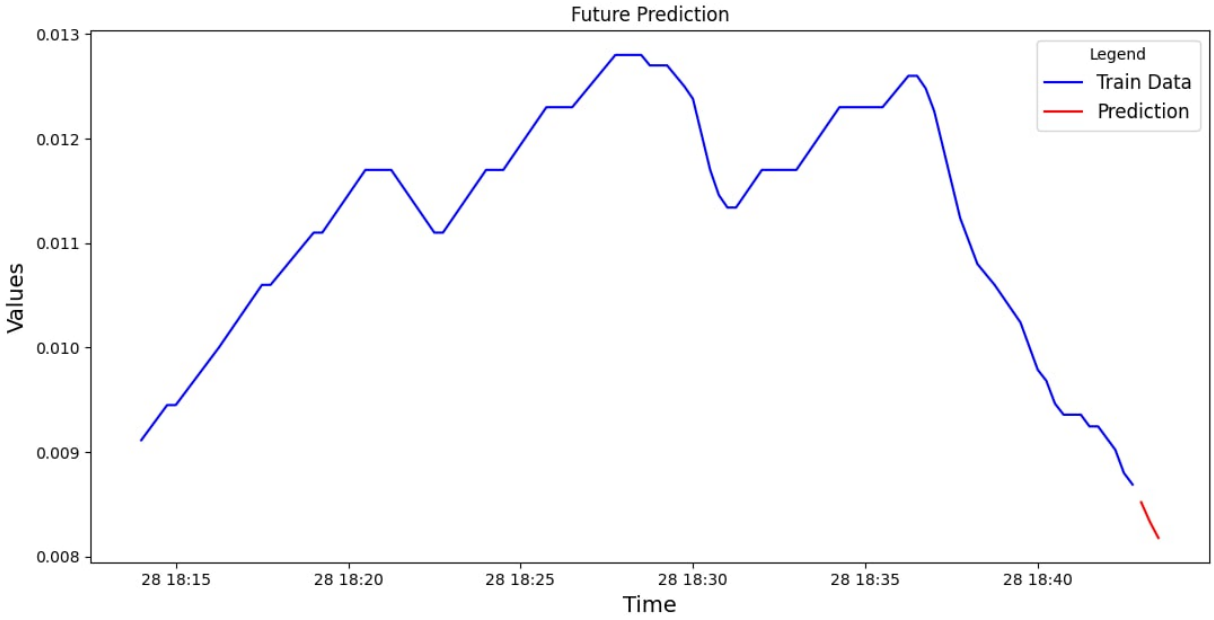

Figura 38: Future Prediction Ordini

4. MENU MICROSERVICE

Tale microservizio consente di gestire - mediante operazioni di inserimento e visualizzazione – i menù.

4.1 STRUTTURA DELLE CLASSI

4.1.1 MODEL

La classe utilizzata per la realizzazione del modello per la gestione del database è la seguente:

```
@Document(collection = "Menu")
public class Menu {

    @Id
    @JsonSerialize(using= ToStringSerializer.class)
    private ObjectId _id;
    private String nome;
    private List<ObjectId> pietanze;
    private String stato;
```

Figura 39: Modello Menu

La classe *Menu* è stata realizzata al fine di definire una struttura costituita dalle pietanze che potrebbero essere ordinate.

4.1.2 CLASSI DI SUPPORTO

Tale classe di supporto è:

- **MenuCreate:** classe realizzata al fine di fornire una struttura per i messaggi inviati sul **topic "menu"**, i quali serviranno per verificare se le pietanze inserite nel menù esistono nel database associato al microservizio.

```
public class MenuCreate implements Serializable {

    private ObjectId menu_id;
    private List<ObjectId> pietanze;
    private String stato;
    private String nome;
```

Figura 40: classe supporto MenuCreate

4.2 SPRING BOOT

4.2.1 CONTROLLER MENU

Il *ControllerMenu* è il componente che applica le funzioni del Controller previste dal pattern MVC con il compito di ricevere le richieste provenienti dal client e di reagire eseguendo operazioni sui dati e quindi sul database.

Sono state implementate le seguenti **API REST**:

- **POST:** permette di aggiungere un menu al database Mongo. Questo è raggiungibile attraverso il **path** "/menu" alla **porta "**8081**"** con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/menu/menu".

```
@PostMapping(path="/menu")
public @ResponseBody String addMenu(@RequestBody Menu menu) throws InterruptedException {
    visitCounter.increment();

    System.out.println(menu.getNome());
    Menu menu1 = repository.findByNome(menu.getNome());

    if(menu1 == null){
        System.out.println( "STAMPA PRIMA DEL SAVE CONTROLLER :" + menu.toString());
        Menu new_menu = repository.save(menu);
        System.out.println( "STAMPA DOPO IL SAVE CONTROLLER :" + new_menu.toString());

        MenuCreate menu_create = new MenuCreate(new_menu.get_id(), menu.getPietanze(), "menu_create", menu.getNome());
        kafkaTemplate.send("menu", "menu_create", new Gson().toJson(menu_create));

        System.out.println("MENU: " + new_menu.get_id());

        Thread.sleep(10000);

        Menu menu_finale = repository.findMenuBy_id(new_menu.get_id());

        if (menu_finale != null && menu_finale.getStato().equals("menu_validate")) {

            menu_finale.setStato("menu_validate");

            repository.save(menu_finale);

            esito = "Menu creato correttamente.";

        } else {

            repository.deleteById(new_menu.get_id());

            esito = "Impossibile creare il menù, non tutte le pietanze inserite sono valide.";

        }

        return esito;
    }else{

        return "Impossibile creare un menù con il nome uguale ad uno già esistente.";
```

Figura 41: POST Menu

- **GET:** permette di visualizzare tutti gli ordini contenuti nel database Mongo. Questo è raggiungibile attraverso il **path** "/menu" alla **porta** "8081" con l'utilizzo di Docker mentre, utilizzando Kubernetes, sarà disponibile all'**end-point** "cluster.dsbd.it/menu/menu".

```
@GetMapping(path="/menu", produces = MediaType.APPLICATION_JSON_VALUE)
public @ResponseBody
ResponseEntity<Object> getAllMenu() {
    visitCounter.increment();

    List<Menu> menu = repository.findAll();

    if (menu.size() != 0) {
        return new ResponseEntity<Object>(menu, HttpStatus.OK);
    } else {
        return new ResponseEntity<Object>("Non sono presenti menù salvati.", HttpStatus.NOT_FOUND);
    }
}
```

Figura 42: GET Menu

4.2.2 SAGA COREOGRAPHY

Anche in questo caso, viene implementato il patter SAGA Coreography per l'inserimento di un nuovo menù nel database. La sua definizione è analoga a quella riportata nel paragrafo $3.2.2.

4.3 MONGO

Per la realizzazione del database è stato utilizzato MongoDB, nonché un database non relazionale. Per raggiungere questo scopo, è stata creata un'interfaccia MenuRepository che estende la MongoRepository. All'interno dell'interfaccia sono stati inseriti i metodi utilizzati al fine di soddisfare le richieste.

```
public interface MenuRepository extends MongoRepository<Menu, ObjectId>{

    public Menu findMenuBy_id(ObjectId _id);

    public Menu findByNome(String nome);

}
```

Figura 43: MenuRepository

Inoltre, al fine di risolvere un'eccezione generata da Spring "Exception opening socket" è stato necessaria la creazione di un ApplicationConfiguration. Si faccia riferimento al paragrafo $2.3.

4.4 DOCKER

4.4.1 DOCKERFILE

Il DockerFile per il microservizio *menu* è così formato:

```
FROM maven:3-jdk-8 AS builder
WORKDIR /project
COPY . .
RUN mvn package


FROM openjdk:8-jdk-alpine

WORKDIR /app
#viene copiato l'artefatto costruito dal builder nell'immagine docker
COPY --from=builder /project/target/MenuMicroservice-0.0.1-SNAPSHOT.jar ./MenuMicroservice.jar
ENTRYPOINT ["/bin/sh", "-c"]
CMD [ "java -jar MenuMicroservice.jar" ]
```

Figura 44: DockerFile menu

Per la sua definizione fare riferimento al paragrafo $2.5.1.

4.4.2 DOCKER-COMPOSE

Il *docker-compose* viene inserito all'interno della cartella complessiva del progetto quindi risulterà essere comune a tutti e tre i microservizi.

Vengono creati i seguenti container:

```yaml
menu-service-db:
  image: mongo
  command: mongod --port 27018
  restart: always
  #environment:
    #MONGO_INITDB_ROOT_USERNAME: root
    #MONGO_INITDB_ROOT_PASSWORD: toor
  ports:
    - 27018:27018
  volumes:
    - mongo-db-data-menu:/data/db

menumicroservice:
  build:
    context: .
    dockerfile: MenuMicroservice/Dockerfile
  ports:
    - 8081:8080
  restart: always
  environment:
    <<:  *mongo-credentials-menu
    KAFKA_GROUP_ID: group-consumer
    KAFKA_ADDRESS: kafka:9092
    KAFKA_TOPIC_ORDERS: "order"
    KAFKA_TOPIC_MENU: "menu"
```

Figura 45: docker-compose Menu

4.5 KUBERNETES

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: menu-service-db
spec:
  selector:
    matchLabels:
      app: menu-service-db
  template:
    metadata:
      labels:
        app: menu-service-db
    spec:
      containers:
      - name: menu-service-db
        image: mongo:latest
        ports:
          - containerPort: 27017
        volumeMounts:
          - mountPath: /data/db
            name: mongo-db-data-menu
      volumes:
      - name: mongo-db-data-menu
        persistentVolumeClaim:
          claimName: mongo-db-data-menu
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: menu-microservice
spec:
  selector:
    matchLabels:
      app: menu-microservice
  template:
    metadata:
      labels:
        app: menu-microservice
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "8080"
        prometheus.io/path: "/actuator/prometheus"
    spec:
      containers:
        - name: menu-microservice
          image: pierluca97/menurepo:latest
          ports:
            - containerPort: 8080
          envFrom:
            - configMapRef:
                name: menu-service-env-file
```

Figura 46: deployment database e microservizio Menu

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: menu-service-env-file
data:
  KAFKA_GROUP_ID: group-consumer-menu
  KAFKA_ADDRESS: kafkabroker:9092
  KAFKA_TOPIC_MENU: "menu"
  MONGODB_HOSTNAME: menu-service-db
  MONGODB_PORT: "27017"
  MONGO_DB_NAME: mydbpmenu
```

Figura 47: Variabili d'ambiente

```yaml
apiVersion: v1
kind: Service
metadata:
  name: menu-microservice
spec:
  ports:
    - protocol: TCP
      name: http-traffic
      port: 8080
      targetPort: 8080
  selector:
    app: menu-microservice
```

Figura 48: Service microservizio

```yaml
apiVersion: v1
kind: Service
metadata:
  name: menu-service-db
spec:
  ports:
    - port: 27017
  selector:
    app: menu-service-db
```

Figura 49: Service database

4.5.1 API GATEWAY

Il pattern API Gateway viene implementato sfruttando il componente di Kubernetes chiamato **Ingress.** Per la sua definizione fare riferiento al paragrafo $2.6.1.

4.5.2 PROMETHEUS E GRAFANA

I risultati ottenuti dopo un monitoraggio di 30 minuti sono i seguenti:

E' stato trovato che la seasonability della serie è 23, i residui sono approssimativamente a media nulla e le predizioni sono abbastanza accurate seppur su un numero ristretto di campioni.

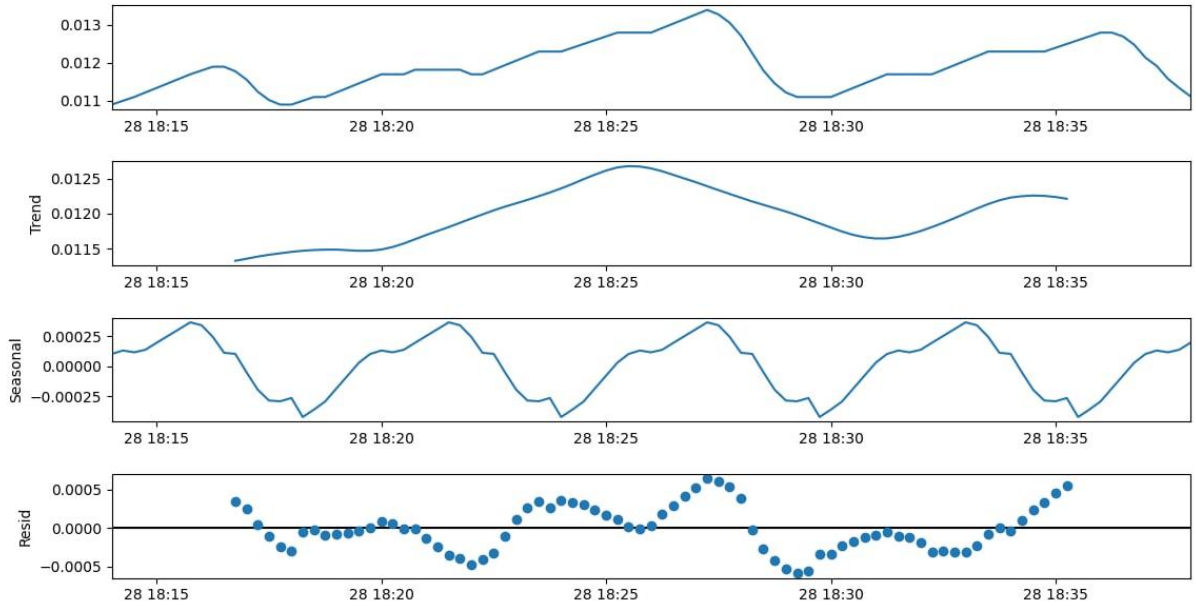

Figura 30: Trend, Stagionalità, Residui Meno

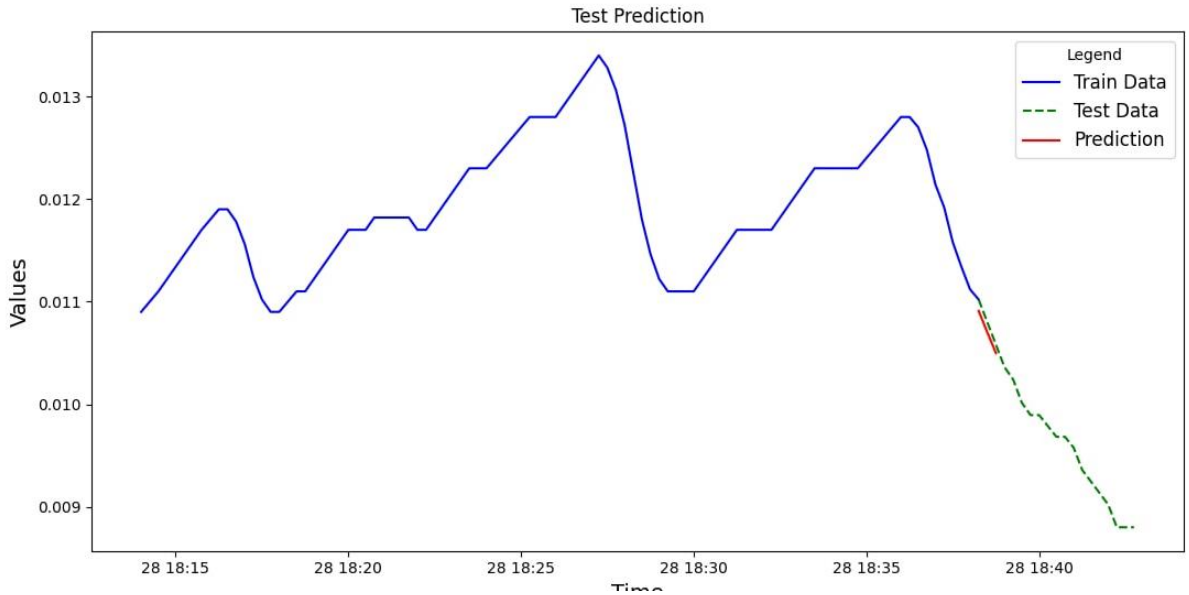

Figura 51: Test Prediction Menu

I risultati sui test data risultano abbastanza precisi e dunque facendo una predizione futura il risultato ottenuto è il seguente:

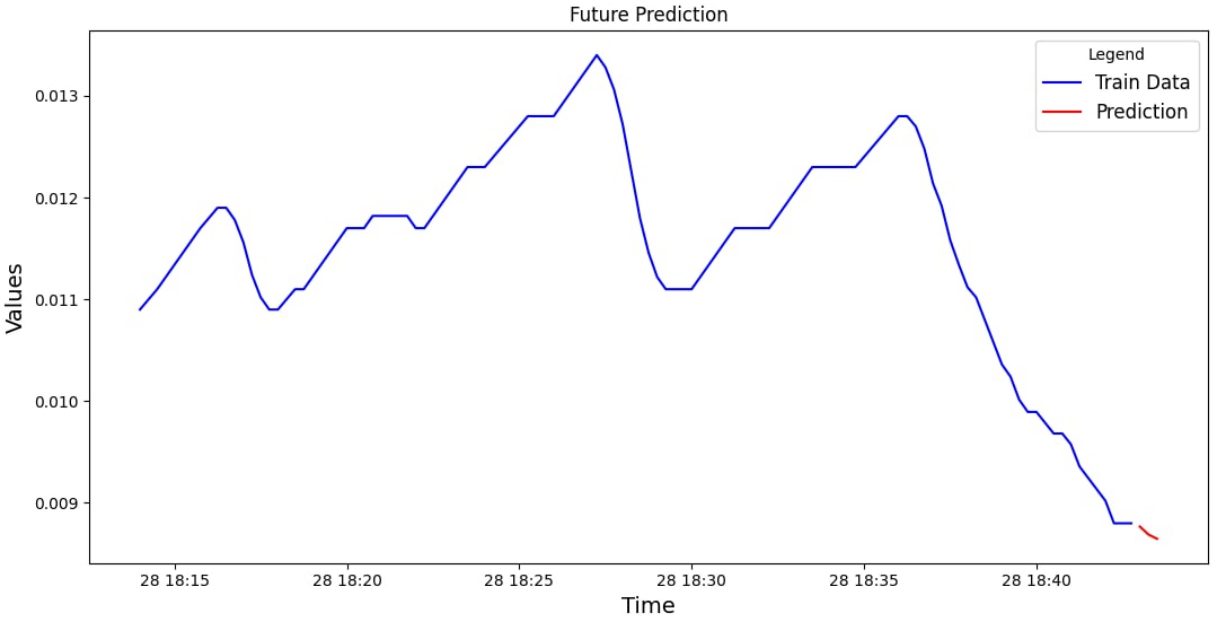

Figura 52: Future Prediction Menu