REACTIVE MANIFESTO

I SISTEMI APPLICATIVI MODERNI DOVREBBERO ADERIRE AI SEGUENTI CANONI:

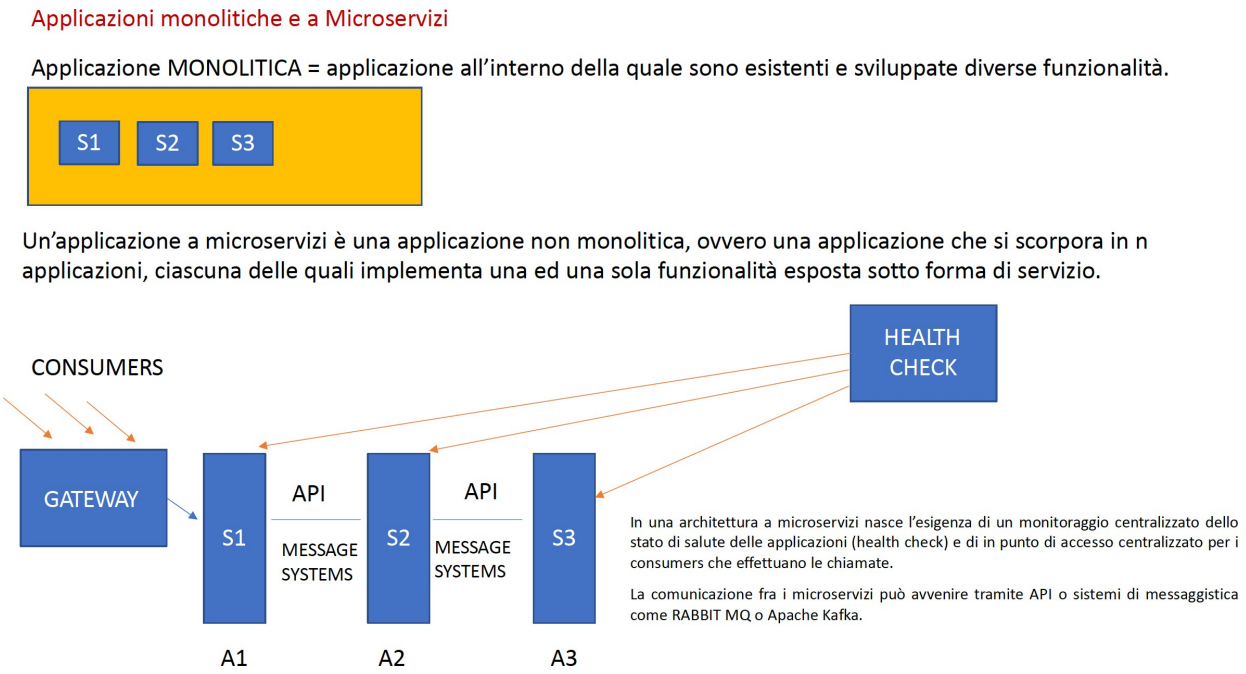
Responsivi: Il sistema, se è in generale possible dare una risposta ai client, la dà in maniera tempestiva. La responsività è la pietra miliare dell'usabilità e dell'utilità del sistema; essa presuppone che i problemi vengano identificati velocemente e gestiti in modo efficace. I sistemi responsivi sono focalizzati a minimizzare il tempo di risposta, individuando per esso un limite massimo prestabilito di modo da garantire una qualità del servizio consistente nel tempo. Il comportamento risultante è quindi predicibile, il che semplifica la gestione delle situazioni di errore, genera fiducia negli utenti finali e predispone ad ulteriori interazioni con il sistema.

Resilienti: Il sistema resta responsivo anche in caso di guasti. Ciò riguarda non solo i sistemi ad alta disponibilità o mission-critical: infatti, accade che ogni sistema che non è resiliente si dimostrerà anche non responsivo in seguito ad un guasto. La resilienza si acquisisce tramite replica, contenimento, isolamento e delega. I guasti sono relegati all'interno di ogni componente, isolando così ogni componente dagli altri e quindi garantendo che il guasto delle singole porzioni del sistema non comprometta il sistema intero. Il recupero di ogni componente viene delegato ad un altro componente (esterno) e l'alta disponibilità viene assicurata tramite replica laddove necessario. I client di un componente vengono dunque alleviati dal compito di gestirne i guasti.

Elastici: Il sistema rimane responsivo sotto carichi di lavoro variabili nel tempo. I Sistemi Reattivi possono adattarsi alle variazioni nella frequenza temporale degli input incrementando o decrementando le risorse allocate al processamento degli stessi. Questo porta ad architetture che non hanno né sezioni contese né colli di bottiglia, favorendo così la distribuibilità o la replica dei componenti e la ripartizione degli input su di essi. I Sistemi Reattivi permettono l'implementazione predittiva, oltre che Reattiva, di algoritmi scalabili perchè fondati sulla misurazione real-time della performance. Tali sistemi, raggiungono l'elasticità in maniera cost-effective su commodity hardware e piattaforme software a basso costo.

Orientati ai Messaggi: I Sistemi Reattivi si basano sullo scambio di messaggi asincrono per delineare per ogni componente il giusto confine che possa garantirne il basso accoppiamento con gli altri, l'isolamento e la trasparenza sul dislocamento e permetta di esprimere i guasti del componente sotto forma di messaggi al fine di delegarne la gestione. L'utilizzo di uno scambio esplicito di messaggi permette migliore gestibilità del carico di lavoro, elasticità e controllo dei flussi di messaggi mediante setup e monitoraggio di code di messaggi all'interno del sistema e mediante l'applicazione di feedback di pressione laddove necessari. Uno scambio di messaggi trasparente rispetto al dislocamento rende possibile, ai fini della gestione dei guasti, l'utilizzo degli stessi costrutti e semantiche sia su cluster che su singoli host. Uno stile di comunicazione non bloccante fa sì che l'entità ricevente possa solo consumare le risorse, il che porta ad un minor sovraccarico sul sistema.

E' UN PENSIERO DIFFUSO CHE LE APPLICAZIONI MONOLITICHE, OVVERO LE APPLICAZIONI IN CUI COESISTONO DIVERSE FUNZIONALITA' ALL'INTERNO DELLO STESSO PACCHETTO APPLICATIVO, ANCHE SE IMPLEMENTATE CON TECNOLOGIE ALL'AVANGUARDIA, DIFFICILMENTE SONO COMPLIANT AL REACTIVE MANIFESTO. E' PER QUESTO MOTIVO CHE SONO NATE LE ARCHITETTURE A MICROSERVIZI; ARCHITETTUTE NELLE QUALI OGNI FUNZIONALITA' VIENE SCORPORATA IN SINGOLO SERVIZIO E IN UN PACCHETTO APPLICATIVO AD ESSA DEDICATO.
IN TAL MODO E' PIU' FACILE LOCALIZZARE E RISOLVERE L'ERRORE, E IL PRINCIPIO DI SINGOLA RESPONSABILITA' VIENE APPLICATO AD OGNI MICROSERVIZIO.



The twelve factory apps

1. **Codebase : ogni Microservizio dovrebbe avere un suo proprio Code Repository**
2. **Dependencies : ogni Microservizio dovrebbe gestire in maniera autonoma le proprie dipendenze**
3. **Config : ogni Microservizio dovrebbe gestire le proprie configurazioni (ad esempio il db)**
4. **Backing Services : il codice non cambia, o cambia pochissimo, se cambia il Repository (DB)**
5. **Build,release,run: ogni Microservizio deve poter essere «deployato» in maniera agevole**
6. **Processes: ogni Microservizio dovrebbe essere Stateless**
7. **Binding: ogni Microservizio dovrebbe essere eseguito su una porta diversa dagli altri**
8. **Concurrency and Balancing: ogni Microservizio dovrebbe essere attivato da un Gateway**
9. **Disposability: il malfunzionamento di un Microservizio non deve inficiare sugli altri**
10. **Dev/Prod Parity: la configurazione applicativa non deve cambiare al cambiare del SO**
11. **Log: ogni Microservizio deve autogestire il Logging**
12. **Admin Processes: ogni Microservizio deve essere monitorabile da un Server**

L'IMPLEMENTAZIONE DI UNA ARCHITETTURA A MICROSERVIZI RICHIEDE IL RISPETTO DI TUTTI I PATTERN SOPRAINDICATI CONOSCIUTI COME TWELVE FACTORY APPS.

REACTIVE MANIFESTO

I SISTEMI APPLICATIVI MODERNI DOVREBBERO ADERIRE AI SEGUENTI CANONI:

Responsivi: Il sistema, se è in generale possible dare una risposta ai client, la dà in maniera tempestiva. La responsività è la pietra miliare dell'usabilità e dell'utilità del sistema; essa presuppone che i problemi vengano identificati velocemente e gestiti in modo efficace. I sistemi responsivi sono focalizzati a minimizzare il tempo di risposta, individuando per esso un limite massimo prestabilito di modo da garantire una qualità del servizio consistente nel tempo. Il comportamento risultante è quindi predicibile, il che semplifica la gestione delle situazioni di errore, genera fiducia negli utenti finali e predispone ad ulteriori interazioni con il sistema.

Resilienti: Il sistema resta responsivo anche in caso di guasti. Ciò riguarda non solo i sistemi ad alta disponibilità o mission-critical: infatti, accade che ogni sistema che non è resiliente si dimostrerà anche non responsivo in seguito ad un guasto. La resilienza si acquisisce tramite replica, contenimento, isolamento e delega. I guasti sono relegati all'interno di ogni componente, isolando così ogni componente dagli altri e quindi garantendo che il guasto delle singole porzioni del sistema non comprometta il sistema intero. Il recupero di ogni componente viene delegato ad un altro componente (esterno) e l'alta disponibilità viene assicurata tramite replica laddove necessario. I client di un componente vengono dunque alleviati dal compito di gestirne i guasti.

Elastici: Il sistema rimane responsivo sotto carichi di lavoro variabili nel tempo. I Sistemi Reattivi possono adattarsi alle variazioni nella frequenza temporale degli input incrementando o decrementando le risorse allocate al processamento degli stessi. Questo porta ad architetture che non hanno né sezioni contese né colli di bottiglia, favorendo così la distribuibilità o la replica dei componenti e la ripartizione degli input su di essi. I Sistemi Reattivi permettono l'implementazione predittiva, oltre che Reattiva, di algoritmi scalabili perchè fondati sulla misurazione real-time della performance. Tali sistemi, raggiungono l'elasticità in maniera cost-effective su commodity hardware e piattaforme software a basso costo.

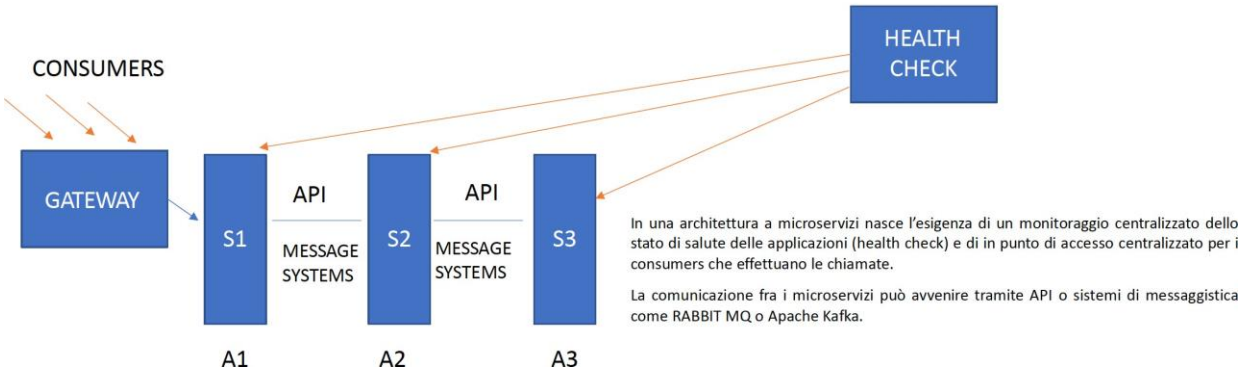
Orientati ai Messaggi: I Sistemi Reattivi si basano sullo scambio di messaggi asincrono per delineare per ogni componente il giusto confine che possa garantirne il basso accoppiamento con gli altri, l'isolamento e la trasparenza sul dislocamento e permetta di esprimere i guasti del componente sotto forma di messaggi al fine di delegarne la gestione. L'utilizzo di uno scambio esplicito di messaggi permette migliore gestibilità del carico di lavoro, elasticità e controllo dei flussi di messaggi mediante setup e monitoraggio di code di messaggi all'interno del sistema e mediante l'applicazione di feedback di pressione laddove necessari. Uno scambio di messaggi trasparente rispetto al dislocamento rende possibile, ai fini della gestione dei guasti, l'utilizzo degli stessi costrutti e semantiche sia su cluster che su singoli host. Uno stile di comunicazione non bloccante fa sì che l'entità ricevente possa solo consumare le risorse, il che porta ad un minor sovraccarico sul sistema.

E’ UN PENSIERO DIFFUSO CHE LE APPLICAZIONI MONOLITICHE, OVVERO LE APPLICAZIONI IN CUI COESISTONO DIVERSE FUNZIONALITA’ ALL’INTERNO DELLO STESSO PACCHETTO APPLICATIVO, ANCHE SE IMPLEMENTATE CON TECNOLOGIE ALL’AVANGUARDIA, DIFFICILMENTE SONO COMPLIANT AL REACTIVE MANIFESTO. E’ PER QUESTO MOTIVO CHE SONO NATE LE ARCHITETTURE A MICROSERVIZI; ARCHITETTUTE NELLE QUALI OGNI FUNZIONALITA’ VIENE SCORPORATA IN SINGOLO SERVIZIO E IN UN PACCHETTO APPLICATIVO AD ESSA DEDICATO.
IN TAL MODO E’ PIU’ FACILE LOCALIZZARE E RISOLVERE L’ERRORE, E IL PRINCIPIO DI SINGOLA RESPONSABILITA’ VIENE APPLICATO AD OGNI MICROSERVIZIO.

Applicazioni monolitiche e a Microservizi

Applicazione MONOLITICA = applicazione all’interno della quale sono esistenti e sviluppate diverse funzionalità.

Un’applicazione a microservizi è una applicazione non monolitica, ovvero una applicazione che si scorpora in n applicazioni, ciascuna delle quali implementa una ed una sola funzionalità esposta sotto forma di servizio.



In una architettura a microservizi nasce l’esigenza di un monitoraggio centralizzato dello stato di salute delle applicazioni (health check) e di in punto di accesso centralizzato per i consumers che effettuano le chiamate.

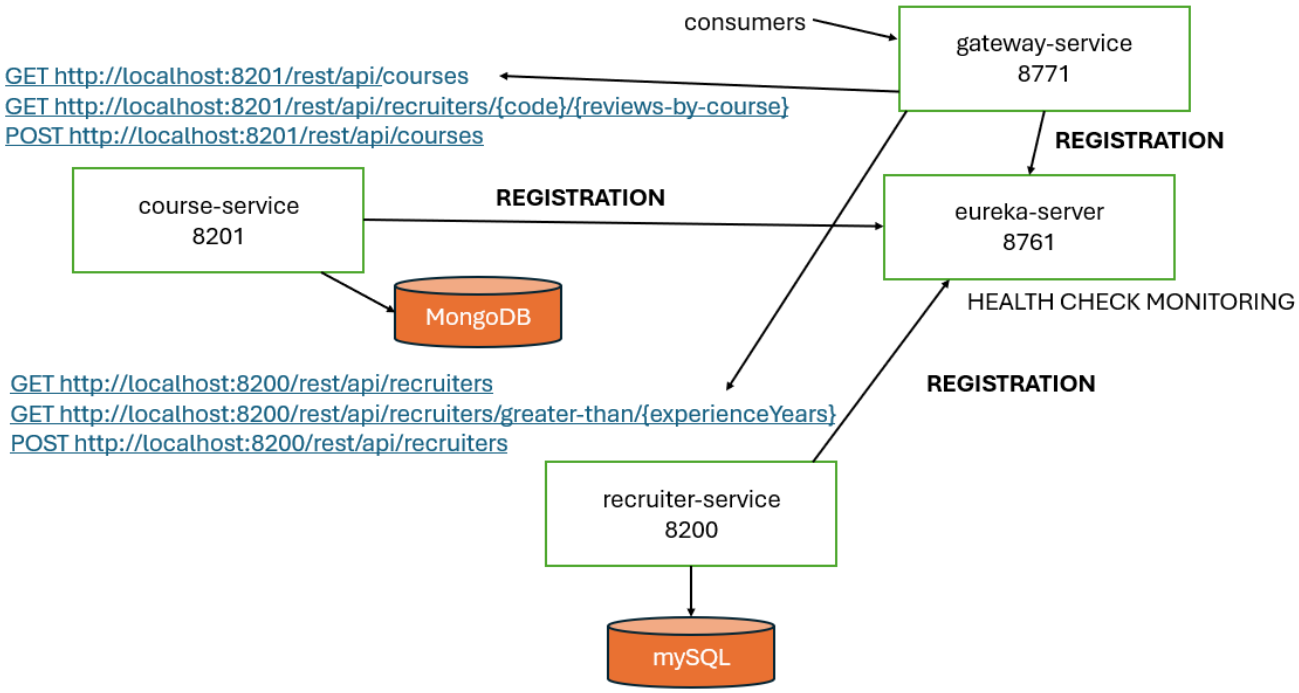
La comunicazione fra i microservizi può avvenire tramite API o sistemi di messaggistica come RABBIT MQ o Apache Kafka.

The twelve factory apps

1. **Codebase : ogni Microservizio dovrebbe avere un suo proprio Code Repository**
2. **Dependencies : ogni Microservizio dovrebbe gestire in maniera autonoma le proprie dipendenze**
3. **Config : ogni Microservizio dovrebbe gestire le proprie configurazioni (ad esempio il db)**
4. **Backing Services : il codice non cambia, o cambia pochissimo, se cambia il Repository (DB)**
5. **Build,release,run: ogni Microservizio deve poter essere «deployato» in maniera agevole**
6. **Processes: ogni Microservizio dovrebbe essere Stateless**
7. **Binding: ogni Microservizio dovrebbe essere eseguito su una porta diversa dagli altri**
8. **Concurrency and Balancing: ogni Microservizio dovrebbe essere attivato da un Gateway**
9. **Disposability: il malfunzionamento di un Microservizio non deve inficiare sugli altri**
10. **Dev/Prod Parity: la configurazione applicativa non deve cambiare al cambiare del SO**
11. **Log: ogni Microservizio deve autogestire il Logging**
12. **Admin Processes: ogni Microservizio deve essere monitorabile da un Server**

L’IMPLEMENTAZIONE DI UNA ARCHITETTURA A MICROSERVIZI RICHIEDE IL RISPETTO DI TUTTI I PATTERN SOPRAINDICATI CONOSCIUTI COME TWELVE FACTORY APPS.

ARCHITETTURA A MICROSERVIZI IMPLEMENTATA CON EUREKA SERVER, GATEWAY E DUE MICROSERVIZI OPERATIVI



CREAZIONE MICROSERVIZIO EUREKA SERVER

1. Dipendenze Gradle: Web, Netflix Eureka Server
2. @EnableEurekaServer in @Spring Boot Application
3. Configurazione application.properties

```
spring.application.name=eureka-server
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

4. Running localhost:8761

CREAZIONE PRIMO MICROSERVIZIO OPERATIVO RECRUITER SERVICE

1. Dipendenze Gradle: Web, Spring Data JPA, mySQL, Netflix Eureka Client
2. @EnableDiscoveryClient in @Spring Boot Application
3. Configurazione application.properties

```
spring.application.name=recruiter-service
server.port=8200
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/openjobmetis
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.generate_statistics=true
logging.level.org.hibernate.SQL=TRACE
logging.level.org.hibernate.stat=TRACE
```

4. Creazione Entity Recruiter

```
@Entity
public class Recruiter implements Serializable{

	private static final long serialVersionUID = 8562914183050874985L;

	@Id
	@Column(name="passport_number")
	private String passportNumber;

	@Column(name="first_name")
	private String firstName;

	@Column(name="last_name")
	private String lastName;

	@Column(name="experience_years")
	private int experienceYears;

	public String getPassportNumber() {
		return passportNumber;
	}

	public void setPassportNumber(String passportNumber) {
		this.passportNumber = passportNumber;
	}

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}
```

```
	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public int getExperienceYears() {
		return experienceYears;
	}

	public void setExperienceYears(int experienceYears) {
		this.experienceYears = experienceYears;
	}

	protected Recruiter() {

	}

	public Recruiter(String passportNumber, String firstName, String lastName, int
experienceYears) {

		this.passportNumber = passportNumber;
		this.firstName = firstName;
		this.lastName = lastName;
		this.experienceYears = experienceYears;
	}


}
```

5. Run per la creazione della tabella

6. Esecuzione dei seguenti script per l'inserimento in tabella

```
INSERT INTO recruiter(passport_number, experience_years, first_name, last_name) VALUES ('1028A',21,'Mario','Rossi');
INSERT INTO recruiter(passport_number, experience_years, first_name, last_name) VALUES ('1028B',12,'Gero','Spalla');
INSERT INTO recruiter(passport_number, experience_years, first_name, last_name) VALUES ('1028C',13,'Erminio','Salvi')
```

7. JpaRepository

```
public interface RecruiterRepository extends JpaRepository<Recruiter,String>{

   public List<Recruiter> findByExperienceYearsGreaterThan(int experienceYears);

}
```

8. Implementazione Service Layer

```
package com.sistemi.informativi.service;

import java.util.List;

import com.sistemi.informativi.entity.Recruiter;

public interface RecruiterService {

	public List<Recruiter> getRecruiters();

	public List<Recruiter> getRecruitersByExperienceYears(int experienceYears);

	public List<Recruiter> saveMoreRecruiters(List<Recruiter> recruiters);



}
```

```
package com.sistemi.informativi.service;

import java.util.List;

import org.springframework.stereotype.Service;

import com.sistemi.informativi.entity.Recruiter;
import com.sistemi.informativi.repository.RecruiterRepository;

@Service
public class RecruiterServiceImpl implements RecruiterService{

	private RecruiterRepository recruiterRepository;

	public RecruiterServiceImpl(RecruiterRepository recruiterRepository) {

		this.recruiterRepository = recruiterRepository;
	}

	// BUSINESS LOGIC TO DO

	@Override
	public List<Recruiter> getRecruiters() {

		return recruiterRepository.findAll();
	}

	@Override
	public List<Recruiter> getRecruitersByExperienceYears(int experienceYears) {

		return recruiterRepository.findByExperienceYearsGreaterThan(experienceYears);
	}
```

```
    @Override
    public List<Recruiter> saveMoreRecruiters(List<Recruiter> recruiters) {
        // saveAll = metodo Spring Data JPA per inserimenti multipli
        return recruiterRepository.saveAll(recruiters);
    }

}
```

9. Implementazione RestController

```
package com.sistemi.informativi.controller;

import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.sistemi.informativi.entity.Recruiter;
import com.sistemi.informativi.service.RecruiterService;

@RestController
@RequestMapping("/rest/api/recruiters")
public class RecruiterController {

    private RecruiterService recruiterService;

    public RecruiterController(RecruiterService recruiterService) {

        this.recruiterService = recruiterService;
    }

    @GetMapping
    public List<Recruiter> getRecruiters(){

        return recruiterService.getRecruiters();
    }


    @GetMapping("/greater-than/{experienceYears}")
    public List<Recruiter> getRecruitersByExperienceYears(@PathVariable ("experienceYears") int
experienceYears){

        return recruiterService.getRecruitersByExperienceYears(experienceYears);

    }

    @PostMapping
    public List<Recruiter> saveMoreRecruiters(@RequestBody List<Recruiter> recruiters){

        return recruiterService.saveMoreRecruiters(recruiters);
```

```
        }

    }
```

10. **Eseguire il RUN dell'applicazione e controllare che sia registrata come client dell'Eureka Server**

11. **Eseguire le seguenti chiamate di test**

GET http://localhost:8200/rest/api/recruiters
GET http://localhost:8200/rest/api/recruiters/greater-than/{experienceYears}
POST http://localhost:8200/rest/api/recruiters/all

Con JSON BODY:

```
[
    {
        "passportNumber": "1028P",
        "firstName": "Gioele",
        "lastName": "Marini",
        "experienceYears": 20
    },
{
        "passportNumber": "1028J",
        "firstName": "Giacomo",
        "lastName": "Pucci",
        "experienceYears": 14
    },
    {
        "passportNumber": "1028G",
        "firstName": "Salvo",
        "lastName": "Picchi",
        "experienceYears": 14
    }
]
```

CREAZIONE SECONDO MICROSERVIZIO OPERATIVO COURSE SERVICE

1. Dipendenze Gradle: Web, Spring Data Mongo DB, Netflix Eureka Client
2. @EnableDiscoveryClient in @Spring Boot Application
3. Configurazione application.properties

```
spring.application.name=course-service
server.port=8201
eureka.client.service-url.default-zone=http://localhost:8761/eureka

spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=openjob
```

4. **Class Review**

```
public class Review {


	private int id;

	private String location;

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getLocation() {
```

```
		return location;
	}

	public void setLocation(String location) {
		this.location = location;
	}

	protected Review() {


	}

	public Review(int id, String location) {

		this.id = id;
		this.location = location;
	}




}
```

5. Class Teacher

```
public class Teacher {

	private String firstName;

	private String lastName;

	private int experienceYears;

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}
```

```
    public int getExperienceYears() {
        return experienceYears;
    }

    public void setExperienceYears(int experienceYears) {
        this.experienceYears = experienceYears;
    }

    protected Teacher() {

    }

    public Teacher(String firstName, String lastName, int experienceYears) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.experienceYears = experienceYears;
    }


    }
```

6. Class Student

```
public class Student {


    private String firstName;

    private  String lastName;

    private int age;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
```

```
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}


	protected Student() {


	}

	public Student(String firstName, String lastName, int age) {

		this.firstName = firstName;
		this.lastName = lastName;
		this.age = age;
	}




	}
```

7. Class Document Course

```
@Document(collection="course")
public class Course {

	@MongoId
	private String code;

	private String title;

	private List<Review> reviews;

	private List<Student> students;

	private List<Teacher> teachers;

	public String getCode() {
		return code;
	}
```

```
	public void setCode(String code) {
		this.code = code;
	}

	public String getTitle() {
		return title;
	}

	public void setTitle(String title) {
		this.title = title;
	}

	public List<Review> getReviews() {
		return reviews;
	}

	public void setReviews(List<Review> reviews) {
		this.reviews = reviews;
	}

	public List<Student> getStudents() {
		return students;
	}

	public void setStudents(List<Student> students) {
		this.students = students;
	}

	public List<Teacher> getTeachers() {
		return teachers;
	}

	public void setTeachers(List<Teacher> teachers) {
		this.teachers = teachers;
	}

	protected Course() {

	}

	public Course(String code, String title, List<Review> reviews, List<Student> students,
List<Teacher> teachers) {

		this.code = code;
		this.title = title;
		this.reviews = reviews;
		this.students = students;
		this.teachers = teachers;
```

```
    }

    }
```

8. Interfaccia CourseRepository

```
public interface CourseRepository extends MongoRepository<Course,String>{
}
```

9. Implementazione Service Layer

```
package com.sistemi.informativi.service;

import java.util.List;

import com.sistemi.informativi.document.Course;
import com.sistemi.informativi.document.Review;

public interface CourseService {

        public Course saveCourse(Course course);

        public List<Course> getCourses();

        public List<Review> getReviewsByCourse(String code) throws Exception;

}


package com.sistemi.informativi.service;

import java.util.List;

import org.springframework.stereotype.Service;

import com.sistemi.informativi.document.Course;
import com.sistemi.informativi.document.Review;
import com.sistemi.informativi.repository.CourseRepository;


@Service
public class CourseServiceImpl implements CourseService{

        private CourseRepository courseRepository;

        public CourseServiceImpl(CourseRepository courseRepository) {

                this.courseRepository = courseRepository;
        }

        // BUSINESS LOGIC TO DO

        @Override
```

```
	public Course saveCourse(Course course) {

		return courseRepository.save(course);
	}

	@Override
	public List<Course> getCourses() {

		return courseRepository.findAll();
	}

	@Override
	public List<Review> getReviewsByCourse(String code) throws Exception {

		return courseRepository.findById(code).
				orElseThrow(()->
				new Exception("course not found")).
				getReviews();

}

}
```

10. Eseguire il run del microservizio e verificare la presenza del microservizio sulla dashboard di Eureka

11. Eseguire le seguenti chiamate Rest

POST http://localhost:8201/rest/api/courses con JSON

```
{
	"code": "1028A",
	"title": "Java",
	"reviews": [
	   {

	    "id":1,
	    "location":"Rome"
	   },
	   {
	    "id":2,
	    "location":"Milan"
	   }
	],
	"teachers": [

	   {
	     "firstName":"Mario",
	     "lastName":"Rossi",
	     "experienceYears":23
	   },
	   {
	     "firstName":"Marco",
	     "lastName":"Verdi",
	     "experienceYears":26
```

```
        }
    ],
    "students": [

        {
          "firstName":"Gioele",
          "lastName":"Marini",
          "age":22
        },
        {
          "firstName":"Gianmarco",
          "lastName":"Bianchi",
          "age":21
        }
    ]


}
```


GET http://localhost:8201/rest/api/courses
GET http://localhost:8201/rest/api/courses/{code}/reviews-by-course

CREAZIONE MICROSERVIZIO GATEWAY

1. Dipendenze Gradle: GATEWAY; EUREKA DISCOVERY CLIENT
2. @EnableDiscoveryClient in @Spring Boot Application
3. Configurazione application.properties

```
spring.application.name=gateway-service
server.port=8771
eureka.client.service-url.default-zone=http://localhost:8761/eureka
spring.cloud.gateway.mvc.routes[0].id=recruiter-service
spring.cloud.gateway.mvc.routes[0].uri=lb://recruiter-service
spring.cloud.gateway.mvc.routes[0].predicates[0]=Path=/rest/api/recruiters/**
spring.cloud.gateway.mvc.routes[1].id=course-service
spring.cloud.gateway.mvc.routes[1].uri=lb://course-service
spring.cloud.gateway.mvc.routes[1].predicates[0]=Path=/rest/api/courses/**
```

4. Test Chiamate REST con POSTMAN

GET http://localhost:8771/rest/api/recruiters
GET http://localhost: 8771/rest/api/recruiters/greater-than/{experienceYears}
POST http://localhost:8771/rest/api/recruiters
POST http://localhost:8771/rest/api/courses
GET http://localhost:8771/rest/api/courses
GET http://localhost:8771/rest/api/courses/{code}/reviews-by-course