Microservizi

- I Microservizi sono servizi che si possono **rilasciare** in modo indipendente e che rappresentano un certo dominio del *business*
- Le funzionalità incluse in un microservizio sono rese accessibili agli altri servizi tramite la rete
- Un microservizio fornisce un **endpoint** che è una coda o una chiamata REST sopra un certo protocollo di comunicazione di rete
- I microservizi abbracciano il concetto di information hiding, ovvero nascondono più informazioni possibile ed espongono il meno possibile attraverso le interfacce
  - Si hanno più tipi di interfacce per la stessa funzionalità, e si usa il simbolo di un esagono per il microservizio

Concetti Chiave dei Microservizi

- **Deploy indipendente**: il rilascio di un microservizio è indipendente dal rilascio degli altri
  - Per permettere il deploy indipendente i microservizi sono accoppiati in modo lasco. I microservizi non condividono i database
- **Modellano un dominio del business** (attività, lavoro)
- **Possiedono il loro stato**: evitano di usare un database condiviso. Se un microservizio ha bisogno dei dati di un altro microservizio li chiede tramite una interfaccia. Questo rende veramente possibile il deploy indipendente
- **Dimensione**: non troppo grande … iniziare con uno o pochi microservizi e quindi aggiungerne altri in modo da rispettare information hiding, e abbassare la complessità

Vantaggi dei Microservizi

- Permettono eterogeneità di tecnologie
- Robustezza
- Scalabilità
- Facilità di deploy
- Componibilità

Stili di Comunicazione fra Microservizi

- Bloccante: un microservizio può effettuare una chiamata a un altro microservizio e aspettare la risposta
- Asincrono: un microservizio può chiamare un altro microservizio e portare avanti altro
- Richiesta-risposta: un microservizio manda una richiesta a un altro microservizio e si aspetta di ricevere una risposta che lo informa del risultato
- Guidato da eventi: i microservizi emettono eventi che altri microservizi consumano e reagiscono di conseguenza. Il microservizio che emette l'evento non conosce quale microservizio consumerà l'evento
- Dati condivisi: i microservizi possono collaborare attraverso una sorgente di dati condivisi

Scelte Tecnologiche

- Alcuni framework permettono di effettuare chiamate a processi remoti come SOAP e gRPC
- REST: è uno stile architetturale attraverso il quale una risorsa (per es. un Ordine o un Cliente) viene esposta e vi si può accedere usando operazioni GET o POST
- GraphQL è un protocollo che permette ai consumatori di definire query che prendono informazioni da microservizi e che filtrano i risultati per restituire ciò che è richiesto
- Message broker: un middleware che permette di effettuare una comunicazione asincrona attraverso code

gRPC

- GRPC è un framework di comunicazione remota sviluppato da Google. Serve a far comunicare microservizi fra loro in modo efficiente
- Usa HTTP/2
- Usa Protocol Buffers (protobuf) invece di JSON
- E' tipizzato

Microservizi con SpringBoot

- Per poter usare i microservizi usando Spring Boot e Visual Studio Code
  - Impostare l'ambiente VS Code installando Spring Boot Extension Pack
  - Creare un progetto Spring Boot usando VS Code Spring Initializr (oppure tramite https://start.spring.io), oppure scaricando un progetto già creato da https://spring.io/guides/gs/spring-boot
- Il progetto userà Spring Web (in modo da avere API REST)

Componenti per gRPC

- Un contratto (protobuf) che definisce il servizio

```
service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}
```

- Una classe che gira su server Spring Boot

```
@GrpcService
public class HelloServiceImpl extends HelloServiceGrpc.HelloServiceImplBase {

    @Override
    public void sayHello(HelloRequest request, StreamObserver<HelloResponse> responseObserver) {
        HelloResponse response = HelloResponse.newBuilder().setMessage("Ciao " + name).build();
        responseObserver.onNext(response);
    }
}
```

- Client Spring Boot

```
@Service
public class HelloClient {
    @GrpcClient("hello-service")
    private HelloServiceGrpc.HelloServiceBlockingStub stub;

    public String call(String name) {
        HelloRequest request = HelloRequest.newBuilder().setName(name).build();
        return stub.sayHello(request).getMessage();
    }
}
```

Sviluppo del Microservizio

- Aprire il file src/main/java/com/example/demo/DemoApplication.java e creare una API REST come segue

```
package com.example.demob;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemobApplication {
  public static void main(String[] args) {
    SpringApplication.run(DemobApplication.class, args);

  }

  @GetMapping("/hello")
  public String hello(@RequestParam(value = "name", defaultValue = "World") String name) {
    return String.format("Hello %s!", name);
  }

  @GetMapping("/helloSpr")
  public String sayHello() {
    return "Hello, Spring Boot!";
  }
}
```


Esecuzione

- Dal terminale si può eseguire mvn spring-boot:run
- Dal browser
  - http://localhost:8080/hello
  - http://localhost:8080/helloSpr
  - http://localhost:8080/hello?name=Emi

Inversion of Control (IoC)

- Spring Boot usa l'inversion of Control (IoC) che è il principio di progettazione secondo cui gli oggetti (istanze) necessarie non si creano implementando codice dell'applicazione ma è il framework Spring Boot a crearle
- La IoC è realizzata tramite lo Spring IoC Container, o ApplicationContext. Il container crea gli oggetti (detti bean), ne gestisce il ciclo di vita e i loro collegamenti
- La Dependency Injection (DI) è la tecnica usata per realizzare la IoC. Con la DI, una classe prende le dipendenze dall'esterno anziché crearle al suo interno

```
@Service
public class OrderService {
    private final OrderRepository orderRepository;

    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }
}

@Repository
public class OrderRepository {
}
```


Dependency Injection

- Spring Boot si serve delle annotazioni @Service e @Repository per individuare le classi utili, riconosce che OrderService necessita di un OrderRepository e gli inietta automaticamente l'istanza corretta
- La dipendenza tramite costruttore (Constructor Injection) è la prassi raccomandata e fornisce: immutabilità, testabilità e dipendenze facili da riconoscere

```
public OrderService(OrderRepository orderRepository) { ...
```

- In alternativa si può usare la Field Injection, annotando un campo

```
@Autowired
private OrderRepository repo;
```

- Oppure la Setter Injection

```
@Autowired
public void setRepo(OrderRepository repo) {
    this.repo = repo;
}
```

Vantaggi Forniti da DI e IoC

- DI e IoC permettono il disaccoppiamento: una classe che dipende da una interfaccia può prendere in input una istanza di una qualunque classe che implementa quell'interfaccia
- Il test è più facile poiché si può passare un mock o uno stub facilmente
- Il codice ha meno dipendenze rigide e quindi è più facile da evolvere
- Quindi, Spring controlla la creazione dei bean e inietta automaticamente le dipendenze

IoC, DI e Dati Persistenti

- I dati di in una tabella del database sono rappresentati nel codice tramite: Entity, Repository, Service, e Controller
- Entity rappresenta il dato; e JPA (Java Persistence API) mappa sulla tabella l'entità

```
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String email;
    private String name;
}
```

- Repository permette l'accesso ai dati della tabella

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);
}
```

- Spring JPA crea la classe concreta, la registra come bean e la rende disponibile. Quindi tramite DI Spring inietta il codice generato

Dependency Injection Su Service

- Service usa il repository iniettato da Spring

```
@Service
public class UserService {

    private final UserRepository userRepository;

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

    public User getUserByEmail(String email) {
        return userRepository
                .findByEmail(email)
                .orElseThrow(() -> new RuntimeException("User not found"));
    }
}
```

- Spring crea il codice di UserRepository che potrebbe usare SQL o altri modi per prendere i dati dal database
- Spring inietta nel costruttore un'istanza di UserRepository

Controller

- Il controller riceve le chiamate dal client e usa Service

```
@RestController
@RequestMapping("/users")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/{email}")
    public User getUser(@PathVariable String email) {
        return userService.getUserByEmail(email);
    }
}
```

- Attraverso le annotazioni RequestMapping e GetMapping si definisce il percorso (path) dell'end point dell'URL, questo viene mappato su un metodo
- Spring inietta l'istanza di UserService al controller e collega tutti gli oggetti che servono

Annotazioni di Spring Boot

- Per i microservizi con Spring Boot si usano varie annotazioni
- @SpringBootApplication indica che la classe è il punto di avvio dell'applicazione (ha il main)
- @Service indica una classe che contiene una business logic
- @RestController indica una classe che è un controller con API REST
- @RequestMapping associa i metodi del controller alle richieste HTTP e definisce i percorsi
- @GetMapping, @PostMapping, @PutMapping, @DeleteMapping sono le annotazioni per le corrispondenti operazioni REST
- @RequestParam prende i parametri dalla richiesta (url)

Annotazioni

- Per dependency injection
  - @AutoWired inietta le dipendenze automaticamente
- Per la persistenza dei dati
  - @Entity indica una classe come una entità JPA (tabella nel database)

Servizi REST

- Un servizio REST (Representational State Transfer) è un servizio su web che è progettato per facilitare la comunicazione di componenti su internet
- Le principali caratteristiche di un servizio REST sono
  - Statelessness: ciascuna richiesta deve contenere le informazioni utili a ricostruire lo stato, poiché il server non conserva informazioni fra una richiesta e la successiva
  - Risorse: le risorse rappresentano qualsiasi dato e ogni risorsa è identificata da un URI (Uniform Resource Identifier) unico
  - Operazioni CRUD: i servizi REST supportano le quattro operazioni base sulle risorse (Creation, Read, Update, Delete). Tali operazioni sono mappate su metodi HTTP come segue: POST per creare, GET per leggere, PUT per aggiornare e DELETE per rimuovere
  - Rappresentazione: le risorse REST sono rappresentate in un formato specifico (spesso JSON o XML). I client richiedono la rappresentazione desiderata nell'header della richiesta

Servizi REST

- Caratteristiche dei servizi REST (continua)
  - Interfaccia Uniforme: i servizi REST hanno un'interfaccia uniforme e consistente, ovvero si usano i metodi Get, Post, Put, Delete di HTTP per interagire con le risorse
  - Comunicazione Stateless: i servizi REST comunicano attraverso un modello richiesta-risposta (request-response) senza stato. Ogni richiesta da un client dovrebbe contenere tutte le informazioni necessarie al server
  - Messaggi auto-descrittivi: le risposte dal server dovrebbero contenere le informazioni per interpretarle, queste informazioni sono contenute negli header HTTP e nei codici di stato
  - Hypermedia Links: gli hyperlink possono essere usati nei servizi REST per permettere ai client di scoprire ulteriori risorse
  - Sistemi a Strati: i servizi REST dovrebbero essere progettati in modo che si possano aggiungere vari intermediari come load balancer, cache, sicurezza