Tutorato Progettazione, Algoritmi e Computabilità

Sviluppo API-based con Spring-Boot

Dott. Michele Beretta

Prof.ssa Patrizia Scandurra

Università degli Studi di Bergamo

2023 – 2024

Outline

API e sviluppo API-based

API

Le **API** (Application Programming Interfaces) sono un insieme definito di regole che contengono metodi ben definiti per la comunicazione tra due software e/o servizi.

- Aiutano software differenti ad interagire tra di loro
- Sono oggi presenti per moltissimi ambiti applicativi
- Devono seguire determinate regole in moto che sia universalmente valido il modo in cui devono essere utilizzate, definite e richiamate

Documentare API

La documentazione di una API copre un ruolo fondamentale: è l'aspetto che determina quanto facilmente una API è utilizzabile.

Una buona documentazione deve contenere:

- Descrizione del servizio offerto
- URL
- Elenco e descrizione di eventuali parametri accettati in input
- Stati HTTP della risposta dalla API

Come realizzare API

Esistono due standard principali per l'implementazione di API:

- REST
- gRPC

REST – Representational State Transfer

REST

REST (Representational State Transfer) è uno stile architetturale per i sistemi distribuiti, con le seguenti caratteristiche

- Stateless
- Sfruttamento dell'architettura fornita da HTTP
- Uso dei verbi HTTP per diverse operazioni

Il funzionamento prevede una struttura degli URL ben definita e la chiamata di questi per l'esecuzione di

- Richieste di informazioni
- Modifiche di dati

Metodo GET

L'operazione di GET permette di leggere il valore di una risorsa.

```
GET /v1/people HTTP/1.1

RISPOSTA
HTTP/1.1 200 OK
Content-Type: application/json
[
{
"id":1,
"firstname": "Dario",
"lastname": "Frongi",
"age":30
},
{
"id":2,
"firstname": "Mario",
"lastname": "Rossi",
"age":30
}
]
```

Metodo POST

L'operazione di POST permette di creare una nuova risorsa sul server.

Nel body della richiesta, di solito, si inseriscono le info dell'oggetto che si vuole creare (senza l'ID, dovrebbe essere assegnato in automatico).

```
POST /v1/people HTTP/1.1
Host: https://api.myapi.it
Content-Type: application/json
{
"firstname": "Dario",
"lastname": "Frongi",
"age":30
}
```

Metodi PUT e PATCH

L'operazione PUT permette di sostituire completamente una risorsa (idempotente).

```
PUT  /v1/people/1 HTTP/1.1
Host: https://api.myapi.it
Content-Type: application/json
{
"firstname": "Mario",
}
```

L'operazione PATCH permette di aggiornare parzialmente una risorsa.

```
PATCH /v1/people/1 HTTP/1.1
Host: https://api.myapi.it
Content-Type: application/json
{
"age": 15,
}
```

Metodo DELETE

L'operazione di `DELETE` permette di cancellare una risorsa (dato un ID).

```
DELETE /v1/people/1
Host: https://api.myapi.com
```

Test di API REST

È possibile usare alcuni software per testare le API:

- HTTPie (`https://github.com/httpie/desktop`)
- Insomnia (`https://insomnia.rest`, richiesto account)
- Postman (`https://www.postman.com`, richiesto account)

REST in Java con Spring

Installazione

`https://spring.io`

È possibile installare **Spring** dal marketplace di Eclipse, oppure seguendo le istruzioni sul sito.

Creazione nuovo progetto (1)

1. Scegliamo un nuovo progetto *Spring Starter Project*
2. Impostiamo
   - Nome
   - Type a *Maven*
3. Aggiungiamo eventuali dipendenze
4. Verrà creato in automatico un file `pom.xml`

Creazione nuovo progetto (2)

Nel file `pom.xml` controlliamo di avere una dipendenza

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

E non

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
```

Spring Boot

Spring Boot è un progetto Spring che ha lo scopo di rendere più semplice lo sviluppo e l'esecuzione di applicazioni Spring, tramite una configurazione automatica (dove possibile) utilizzando dei valori di default scelti ad hoc.

Una maggiore semplicità è data anche dal fatto che Spring Boot incapsula dentro un JAR un'istanza dell'application server TomCat.

Fornisce anche delle opzioni per il build ed il deploy delle applicazioni in produzione.

DEMO

Vediamo insieme la struttura di una applicazione Spring Boot per la gestione di un ristorante.

Si vuole definire un servizio applicativo `restaurant-service` per la gestione di un insieme di ristoranti:

- I ristoranti sono definiti come oggetti di tipo `Restaurant`
- I ristoranti vengono memorizzati su di un database (per semplicità usiamo un `ArrayList`)
- Viene definito un servizio `RestaurantService`

@SpringBootApplication

La classe che viene annotata con `@SpringBootApplication` ha come unico scopo quello di lanciare l'applicazione Spring.

```java
@SpringBootApplication
public class Tutorato6RestaurantApplication {

    public static void main(String[] args) {
        SpringApplication.run(Tutorato6RestaurantApplication.class, args);
    }

}
```

Tramite questa annotazione, Spring Boot capisce che questa classe deve essere usata per l'avvio dell'applicazione e crea l'application context.

application.properties

Le applicazioni Spring Boot possono essere configurate mediante dei file di proprietà in cui è possibile specificare sia le proprietà comuni di Spring Boot che le proprietà specifiche dell'applicazione.

Se una certa proprietà non viene specificata nel file, viene usato il valore di default (ad esempio 8080 per la porta):

```
# application.properties
server.port = 8080
```

@Component

Con l'annotazione @Component possiamo definire del codice da eseguire insieme all'applicazione, ad esempio per effettuare log oppure inizializzazioni/inserimenti iniziali per avere dei dati di prova.

```
@Component
public class InitRestaurantDb implements CommandLineRunner {

    @Autowired
    private RestaurantService restaurantService;

    public void run(String[] args) {
        restaurantService.createRestaurant( "Hostaria dell'Orso", "Roma" );
        restaurantService.createRestaurant( "Baffetto", "Roma" );
        restaurantService.createRestaurant( "L'Omo", "Roma" );
        restaurantService.createRestaurant( "Seta", "Milano" );
    }

}
```

`@Component`

Da notare:

- Un componente *deve* avere un metodo `void run(String[] args)`
- L'annotazione `@Autowired` definisce un campo che dev'essere "collegato" prima dell'esecuzione di qualsiasi altro codice

Entità d'interesse (*Presentation Model*)

L'obiettivo è quello di implementare un servizio che gestisca determinate entità. Il formato con cui si comunica su REST è solitamente JSON. Dobbiamo quindi dichiarare la classe che conterrà il nostro ristorante, e che conterrà anche i risultati delle esecuzioni del servizio REST.

```java
public class Restaurant {

    private static long N_RESTAURANT = 0;

    private Long id;
    private String name;
    private String location;

    public Restaurant(String name, String location) {
        this.id = N_RESTAURANT;
        this.name = name;
        this.location = location;
        N_RESTAURANT++;
    }

    public String getName() {
        return this.name;
    }

    public String getLocation() {
        return this.location;
    }

}
```

Entità d'interesse (*Presentation Model*)

Se avessimo un database, questa classe potrebbe essere annotata con `@Entity`. In tal modo, il get/set di oggetti `Restaurant` sarebbe transparentemente riportato su database (*persistenza*).

```
public class Restaurant {

    private static long N_RESTAURANT = 0;

    private Long id;
    private String name;
    private String location;

    public Restaurant(String name, String location) {
        this.id = N_RESTAURANT;
        this.name = name;
        this.location = location;
        N_RESTAURANT++;
    }

    public String getName() {
        return this.name;
    }

    public String getLocation() {
        return this.location;
    }

}
```

RestaurantRepository

Creiamo questa classe per fare in modo di memorizzare tutti i ristoranti che ci servono. In una applicazione vera, questo repository si interfaccerebbe direttamente con un database per avere la persistenza.

```
public class RestaurantRepository {

    ArrayList<Restaurant> listaRistoranti = new ArrayList<Restaurant>();

    public Restaurant save(Restaurant restaurant) {
        this.listaRistoranti.add(restaurant);
        return restaurant;
    }

    public Collection<Restaurant> findAll() {
        return listaRistoranti;
    }

    public Restaurant findByName(String name) {
        for (Restaurant r : listaRistoranti) {
            if (r.getName().equals(name))
                return r;
        }
```

`@Service` (*Domain Model*)

Il passo successivo è quello di definire una classe che implementi il servizio remoto REST. Un servizio rappresenta un'attività, un processo o una trasformazione significativa nel dominio, la cui responsabilità non può essere assegnata in modo naturale ad una singola entità.

Nel nostro caso, noi vogliamo gestire i ristoranti, quindi avremo bisogno di un servizio che si occupi di inserirli/cercarli.

```java
@Service
public class RestaurantService {

    private RestaurantRepository restaurantRepository = new RestaurantRepository();

    public Restaurant createRestaurant(String name, String location) {
        Restaurant restaurant = new Restaurant(name, location);
        restaurant = restaurantRepository.save(restaurant);
        return restaurant;
    }

    public Restaurant getRestaurant(Long id) {
        Restaurant restaurant = restaurantRepository.findById(id);
        return restaurant;
    }

    public Restaurant getRestaurantByName(String name) {
        Restaurant restaurant = restaurantRepository.findByName(name);
        return restaurant;
    }
```

@RestController

Il `@RestController` è la classe che si occupa di gestire le richieste REST e di richiamare il servizio richiesto. Al suo interno, il controller può avere diversi metodi, ciascuno dei quali può svolgere una funzione differente e rispondere a URI differenti, con metodi differenti (POST, GET, etc).

```java
@RestController
public class RestaurantWebController {

    @Autowired
    private RestaurantService restaurantService;

    /* Trova il ristorante con restaurantId. */
    @GetMapping("/restaurants/{restaurantId}")
    public Restaurant getRestaurant(@PathVariable Long restaurantId) {
        Restaurant restaurant = restaurantService.getRestaurant(restaurantId);
        return restaurant;
    }
```

Da notare: `@Autowired`, `@GetMapping`, `@PathVariable`.

@Rest Controller

È anche possibile specificare il valore di default di un parametro in caso non venga specifcato nella chiamata REST.

```
/* Crea un nuovo ristorante. */
@PostMapping("/restaurants")
public Restaurant addRestaurant(@RequestParam(defaultValue = "") String name,
        @RequestParam String location) {
    Restaurant restaurant = restaurantService.createRestaurant(name, location);
    return restaurant;
}
```

Questo endpoint può essere chiamato con

```
POST localhost:8080/restaurants
```

E con `name=Prova&location=Roma` nel corpo.

Jackson Mapper

Utilizzando Spring Boot, automaticamente viene anche incluso il mapper *Jackson*.

Utilizzando l'annotazione `@GetMapping` (o `@PostMapping` e così via) Jackson capisce che se il metodo deve ritornare qualcosa, questo deve essere convertito in automatico in JSON.

REST client con Spring Boot

Con Spring Boot è possibile anche creare un client REST, non solo un server.

1. Creiamo un progetto Spring esattamente come fatto in precedenza
2. Creiamo una classe per il parsing del JSON (o usare la stessa che abbiamo nel controller, copiandola o includendo il progetto come dipendenza)
3. In `application.properties` specifichiamo una nuova porta (e.g. 8081)
4. Nel file con la `@SpringBootApplication` aggiungiamo due metodi annotati con `@Bean` (i.e. gestiti da Spring in automatico)

```
@Bean
RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}

@Bean
CommandLineRunner run(RestTemplate restTemplate) {
    return args -> {
        ArrayList<Restaurant> list = restTemplate.getForObject(
                "http://localhost:8080/restaurants", ArrayList.class);
        System.out.println(list);
    };
}
```

Il metodo `run` gestisce le chiamate, il suo corpo effettua la chiamata.

REST client con Spring Boot

Eseguento il client come Spring Boot application si hanno i seguenti log.

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.5)

2022-11-15 12:02:37.384  INFO 93227 --- [           main] r.c.Tutorato6RestaurantClientApplication : Starting Tutora
2022-11-15 12:02:37.386  INFO 93227 --- [           main] r.c.Tutorato6RestaurantClientApplication : No active profi
2022-11-15 12:02:37.798  INFO 93227 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initiali
2022-11-15 12:02:37.803  INFO 93227 --- [           main] o.apache.catalina.core.StandardService   : Starting servic
2022-11-15 12:02:37.803  INFO 93227 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servle
2022-11-15 12:02:37.840  INFO 93227 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Sp
2022-11-15 12:02:37.840  INFO 93227 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplica
2022-11-15 12:02:38.017  INFO 93227 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started
2022-11-15 12:02:38.022  INFO 93227 --- [           main] r.c.Tutorato6RestaurantClientApplication : Started Tutorat
[{id=0, name=Hostaria dell'Orso, location=Roma}, {id=1, name=Baffetto, location=Roma}, {id=2, name=L'Omo, location=R
2022-11-15 12:05:59.435  INFO 93227 --- [n(12)-127.0.0.1] inMXBeanRegistrar$SpringApplicationAdmin : Application shu
2022-11-15 12:05:59.456  INFO 93227 --- [n(12)-127.0.0.1] o.apache.catalina.core.StandardService   : Stopping servic
```

Notare la stampa dei ristoranti alla terzultima riga.

Esercizi

Esercizio 1

Al progetto visto fino ad ora per il client, aggiungere le chiamate a tutti i metodi che sono offerti dal servizio RestaurantService.

Esercizio 2

Al progetto visto fino ad ora aggiungere:

- La possibilità di ricercare tutti i ristoranti che hanno il nome che inizia per una certa stringa
- La possibilità di eliminare un ristorante dato il suo nome

Provare a far eseguire delle chiamate a queste API da un client realizzato come mostrato in precedenza.

Esercizio 3

Al progetto visto fino ad ora aggiungere, in un servizio separato:

- La possibilità di gestire un menù per ogni ristorante (con aggiunta diretta di tutto il menù o portata per portata)
- Un metodo che dato il nome del ristorante ritorna il suo menù

Provare a far eseguire delle chiamate a queste API da un client realizzato come mostrato in precedenza.

Esercizio 4

Sfruttando le APIs definite nei punti precedenti, realizzare un programma Java che, dato il nome di un ristorante chiesto all'utente, visualizzi il menù.