Luca Cabibbo

Architettura dei Sistemi Software

Invocazione remota: REST

dispensa asw835

ottobre 2024

As we all should know,
REST is not the answer to all questions.
Martin Fowler

- Riferimenti

- Luca Cabibbo. **Architettura del Software**: Strutture e Qualità. Edizioni Efesto, 2021.
  - Capitolo 23, Invocazione remota
- Richardson, C. **Microservices Patterns**: With examples in Java. Manning, 2019.
  - Chapter 3, Interprocess communication in a microservice architecture
- Walls, C. **Spring in Action**, sixth edition. Manning, 2022.
  - Chapter 7, Creating REST services
- JSON
  - https://www.json.org/

- Obiettivi e argomenti

- Obiettivi
  - presentare l'invocazione remota tramite REST
- Argomenti
  - introduzione a REST
  - esempi
  - invocazioni remote asincrone
  - discussione

* Introduzione a REST

- Il termine *REST* indica diverse cose
  - REST è uno stile architetturale per descrivere l'architettura del world-wide web moderno, e per guidare la progettazione e l'implementazione di sistemi distribuiti sul web
    - ci occuperemo di questo in una successiva dispensa
  - REST è, in pratica, una tecnologia per l'invocazione remota, basata su HTTP
    - l'argomento di questa dispensa

REST

- In questa dispensa, **REST** (*REpresentational State Transfer*) indica una tecnologia per l'invocazione remota, basata su HTTP
  - in questa accezione, REST consente di
    - (lato server) associare un'operazione del server con un metodo HTTP relativa a un certo URI
    - (lato client) effettuare l'invocazione remota di un'operazione del server – tramite una richiesta HTTP all'URI associato all'operazione
  - queste operazioni remote possono anche avere parametri e restituire risultati
    - sia i parametri che i risultati possono essere valori atomici oppure oggetti con una struttura complessa
    - questi dati possono essere scambiati su HTTP in diversi modi – ad es., mediante parametri nel path oppure nel corpo mediante JSON o XML

REST e Spring

- Spring Web MVC fornisce anche il supporto per REST
  - è sufficiente annotare una classe con **@RestController**
    - in alternativa, è possibile usare l'annotazione **@Controller** e poi annotare i parametri delle operazioni remote con **@RequestBody** e il risultato con **@ResponseBody**
  - sia i parametri che il risultato di un'operazione remota possono essere "atomici" (come interi e stringhe) oppure anche altri oggetti – oggetti del dominio o, meglio, oggetti di un opportuno Presentation Model
    - questi dati vengono scambiati su HTTP – in genere mediante JSON o anche altri formati
    - Spring si può occupare della conversione di dati nel o nei formati di interscambio di interesse in modo trasparente allo sviluppatore – o comunque in modo semplificato

Risorse

- Uno dei concetti centrali di REST è quello di “risorsa”
  - una *risorsa* (*resource*) è ogni elemento informativo di interesse a cui può essere attribuito un nome
    - ad es., il “corso di Architettura dei Sistemi Software a Roma Tre” oppure “il tempo a Roma oggi”
  - un *identificatore di risorsa* (*resource identifier*) è un nome univoco usato per identificare una specifica risorsa – ad es., un URI (Uniform Resource Identifier)
    - ad es., l’URI http://www.uniroma3.it/corsi/asw per la risorsa “corso di Architettura dei Sistemi Software a Roma Tre”

Rappresentazioni

- Un altro concetto centrale di REST è quello di “rappresentazione”
  - una *rappresentazione* è un gruppo di dati (e metadati) per una risorsa (di solito auto-descrittivo)
    - ad es., un documento JSON oppure XML
  - nella comunicazione tra un client e un servizio, vengono scambiate rappresentazioni di risorse – e non risorse
    - ad es., quando un client accede a una risorsa, non gli viene restituita la risorsa, ma piuttosto una sua rappresentazione
    - l’uso delle rappresentazioni consente di disaccoppiare il modo in cui i servizi memorizzano internamente le risorse dal modo (o dai modi) con cui le condividono esternamente

Servizi REST

- In genere, un *servizio REST* consente di gestire una o più collezioni omogenee di risorse
  - ad es., un insieme di corsi e un insieme di docenti
  - il servizio definisce, per ciascuna collezione, un URI di base – chiamato *collection URI*
    - ad es., http://www.uniroma3.it/corsi e http://www.uniroma3.it/docenti
  - ciascuna istanza di risorsa ha un proprio URI – *element URI*
    - ad es., http://www.uniroma3.it/corsi/asw e http://www.uniroma3.it/docenti/luca.cabibbo
  - le operazioni offerte dal servizio sono messe in corrispondenza con i metodi HTTP – come GET, PUT, POST, PATCH e DELETE – sulla base di un'interfaccia uniforme
  - le risorse vengono scambiate mediante una loro rappresentazione – ad es., in formato JSON

Servizi REST

- Ecco le operazioni comunemente definite da un servizio REST
  - operazioni riferite a un collection URI
    - GET – restituisce tutti gli elementi della collezione
    - PUT – sostituisce la collezione con un'altra collezione
    - POST – crea un nuovo elemento della collezione, e gli assegna un nuovo URI
    - PATCH – modifica la collezione sulla base di una lista di cambiamenti specificati nella richiesta
    - DELETE – cancella l'intera collezione

Servizi REST

- Ecco le operazioni comunemente definite da un servizio REST
  - operazioni riferite a un element URI
    - GET – restituisce uno specifico elemento della collezione
    - PUT – crea un nuovo elemento della collezione, oppure lo aggiorna
    - POST – considera l'elemento della collezione come un'altra collezione, e ne aggiunge un elemento
    - PATCH – aggiorna parzialmente uno specifico elemento della collezione
    - DELETE – cancella l'elemento della collezione

JSON

- **JSON** (*JavaScript Object Notation*) è un formato di interscambio di dati, testuale e leggero – facile per le persone da leggere e scrivere – e, soprattutto, facile per il software da interpretare e da generare (in una varietà di linguaggi di programmazione)
  - due costrutti principali
    - un *oggetto* (un record) è un gruppo di coppie nome/valore
    - un *array* (una lista) è una sequenza ordinata di valori
  - i valori possono essere stringhe, numeri, valori booleani, null – oppure (ricorsivamente) anche oggetti o array
  - i nomi sono delle stringhe

JSON

- **JSON** (*JavaScript Object Notation*) è un formato di interscambio di dati, testuale e leggero – facile per le persone da leggere e scrivere – e, soprattutto, facile per il software da interpretare e da generare (in una varietà di linguaggi di programmazione)
  - un esempio di oggetto

```
{
   "employeeId" : 42,
   "firstName" : "John",
   "lastName" : "Doe"
}
```

  - un esempio di oggetto contenente un array di oggetti

```
{
   "employees" : [
      { "firstName" : "John", "lastName" : "Doe" },
      { "firstName" : "Anna", "lastName" : "Smith" }
   ]
}
```


* Esempi

- Vengono ora mostrati alcuni esempi di utilizzo di REST
  - un semplice servizio per saluti
  - il servizio **restaurant-service** per la gestione di un insieme di ristoranti – nell'ambito di un'applicazione **efood** per la gestione di un servizio di ordinazione e spedizione a domicilio di pasti da ristoranti, su scala nazionale

- Il servizio Hello

- Si consideri un semplice servizio per generare dei saluti, la cui logica di business è definita come segue

```
package asw.hello.domain;

import org.springframework.stereotype.Service;

@Service
public class HelloService {

    public String sayHello(String name) {
        return "Hello, " + name + "!";
    }

}
```

  - vogliamo esporre questo servizio come un servizio remoto REST

- Definizione dell'interfaccia del servizio

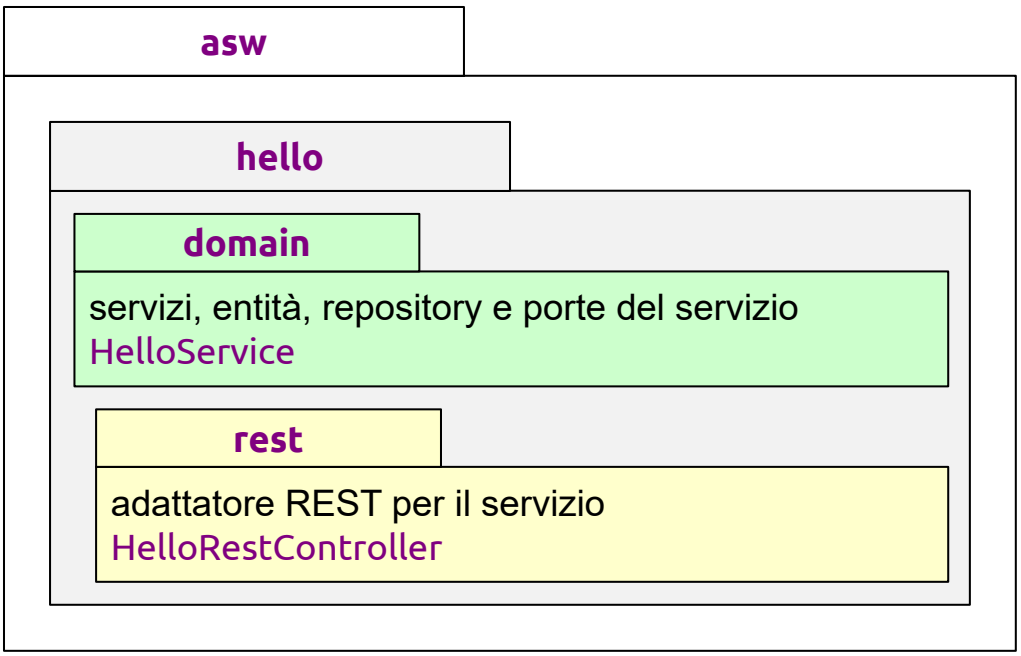
- Con REST, non è necessario specificare l'interfaccia di un servizio in modo formale o con un IDL – anche perché, attualmente, per tale scopo non ci sono ancora degli standard accettati in modo diffuso
  - in ogni caso, l'interfaccia del servizio va comunque stabilita – per ogni operazione del servizio bisogna definire almeno
    - il metodo HTTP e il path
    - i parametri, con i loro tipi – e se vengono scambiati nel path o nel corpo
    - i risultati, con i loro tipi
  - operazione sayHello
    - GET /hello/{name}
    - il parametro name è una stringa, nel path
    - il risultato è una stringa

- Server REST

- Lato server, bisogna definire un server REST per il servizio
  - lo realizziamo come un'applicazione Spring Boot
  - applicando l'architettura esagonale, dobbiamo definire un inbound adapter per questo servizio
    - a tal fine, definiamo il package asw.hello.rest con una classe HelloRestController che implementa il controller REST di Spring Web MVC (ovvero, l'adattatore inbound REST) per il servizio Hello

Architettura esagonale

- Architettura esagonale del servizio **hello**



Server REST

- Lato server, la classe HelloRestController è il controller REST per il servizio

```
package asw.hello.rest;

import asw.hello.domain.HelloService;

import org.springframework.web.bind.annotation.*;

import org.springframework.beans.factory.annotation.Autowired;

@RestController
public class HelloRestController {

   @Autowired
   private HelloService helloService;

   ... vedi dopo ...

}
```

in **rosso** indichiamo il codice relativo a REST

  - l'annotazione **@RestController** indica un tipo di **@Component** – un controller REST di Spring Web MVC

Server REST

- Il metodo sayHello di HelloRestController realizza il collegamento tra il controller REST e il servizio helloService

```
/* Restituisce un saluto a "name".
 * Operazione acceduta come GET /hello/{name} */
@GetMapping("/hello/{name}")
public String sayHello(@PathVariable String name) {

   return helloService.sayHello(name);

}
```

  - l'annotazione **@GetMapping** associa un'operazione al metodo HTTP GET, per il path specificato (nell'esempio, /hello/*{name}*)
  - l'annotazione **@PathVariable** indica una parte variabile del path, che il controller riceve mediante un parametro
  - l'operazione restituisce una stringa – ad es., una richiesta GET al path /hello/World restituisce la stringa Hello, World!

Server REST

- Il controller REST HelloRestController è l'adattatore lato server
  - il ruolo di un adattatore inbound è quello di interpretare richieste del client (in questo caso, richieste REST), trasformarle in richieste all'oggetto adattato (in questo caso, il servizio helloService), ottenere risposte dall'adattato, trasformarle in risposte al client
  - si noti pertanto l'adattamento svolto dal metodo sayHello per gestire una chiamata tramite REST (con il supporto di Spring Web MVC)
    - specifica come catturare l'invocazione mediante l'annotazione **@GetMapping**
    - estrae i parametri dell'invocazione tramite l'annotazione **@PathVariable**
    - invoca l'operazione richiesta del servizio helloService e ottiene il risultato
    - restituisce un risultato – che Spring trasmetterà al client

- Client REST

- Come client per questo servizio è possibile usare un qualunque client HTTP
  - ad es., un browser web, **curl** oppure **Postman**
  - ad esempio
    - curl -s --get "http://localhost:8080/hello/World"
  - tuttavia, qui siamo interessati ad accedere a questo servizio da parte di un'altra applicazione o servizio applicativo
    - questo avviene in genere mediante l'utilizzo di una delle tante librerie per la realizzazione di client HTTP – ad es., **RestTemplate** o **WebClient**

Client REST

- Il lato client è relativo a un'altra applicazione o servizio che vuole accedere, in questo esempio, al servizio Hello
  - supponiamo che sia un'altra applicazione Spring Boot, il cui package di base è asw.samplehelloclient
  - nel suo dominio definiamo un'interfaccia richiesta (una porta) HelloClientPort per accedere al servizio Hello

```
package asw.samplehelloclient.domain;

public interface HelloClientPort {

   public String sayHello(String name);

}
```

  - applicando l'architettura esagonale, va definito un outbound adapter per accedere al servizio Hello, che implementa questa porta/interfaccia
    - definiamo il package asw.samplehelloclient.helloclient.rest con la classe HelloClientRestAdapter

Architettura esagonale

- Architettura esagonale del servizio **sample-hello-client**

Client REST

- Che cosa faccia il client finale di questo servizio è, in effetti, poco rilevante per la nostra discussione
  - tuttavia, ecco una porzione di esempio del nostro client – si noti la dipendenza dalla sola porta HelloClientPort – e soprattutto che il client non dipende in alcun modo da REST

```
package asw.samplehelloclient.domain;

import ...;

@Component
public class HelloClientRunner implements CommandLineRunner {

    @Autowired
    private HelloClientPort helloClientAdapter;

    public void run(String[] args) {

        ... helloClientAdapter.sayHello("Luca") ...
        ... helloClientAdapter.sayHello("World") ...

    }

}
```


Client REST

- Lato client, la classe HelloClientRestAdapter implementa l'adattatore REST per l'accesso al servizio Hello

```
package asw.samplehelloclient.helloclient.rest;

import asw.samplehelloclient.domain.HelloClientPort;

import org.springframework.web.reactive.function.client.*;

import reactor.core.publisher.Mono;

import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Value;

@Service
public class HelloClientRestAdapter implements HelloClientPort {

    ... vedi dopo ...

}
```

Client REST

- Anche nella classe HelloClientRestAdapter sono necessarie alcune definizioni preliminari

```
@Value("${asw.helloservice.rest.uri}")
private String helloServiceUri;
```

```
# application.properties
asw.helloservice.rest.uri=http://localhost:8080
```

```
private WebClient webClient;

public HelloClientRestAdapter(WebClient.Builder webClientBuilder) {
    this.webClient =
          webClientBuilder.baseUrl(helloServiceUri).build();
}
```

  - **WebClient** è un client reactive, non bloccante, per effettuare richieste HTTP
    - va usata la dipendenza starter spring-boot-starter-webflux
  - un'alternativa (meno moderna e sconsigliata) è **RestTemplate**

Client REST

- Ecco il metodo sayHello, che implementa l'accesso all'operazione remota

```
public String sayHello(String name) {

    String greeting = null;
    Mono<String> response = webClient
                        .get()
                        .uri(helloServiceUri + "/hello/{name}", name)
                        .retrieve()
                        .bodyToMono(String.class);    // non bloccante
    try {
        greeting = response.block();    // bloccante
    } catch (WebClientException e) {
        ... REST call failed ...
    }
    return greeting;

}
```

Client REST

- Il ruolo di un adattatore outbound è quello di interpretare richieste della logica di business tramite una porta inbound (in questo caso, richieste Java), trasformarle in richieste all'entità esterna adattata (in questo caso, richieste REST al servizio **hello**), ottenere risposte dall'entità esterna adattata, trasformarle in risposte alla logica di business
  - si noti pertanto l'adattamento svolto dal metodo sayHello per effettuare una chiamata tramite REST
    - in questo caso viene utilizzato il supporto di un **WebClient** – che è una classe "template" per effettuare chiamate HTTP in modo semplice

Client REST (RestTemplate)

- In alternativa, ecco un client REST basato su **RestTemplate**
  - va usata la dipendenza starter spring-boot-starter-web
  - definizioni preliminari

```
@Value("${asw.helloservice.rest.uri}")
private String helloServiceUri;
```

```
# application.properties
asw.helloservice.rest.uri=http://localhost:8080
```

```
private RestTemplate restTemplate = new RestTemplate();
```

Client REST (RestTemplate)

- In alternativa, ecco un client REST basato su **RestTemplate**
  - il metodo che implementa l'accesso all'operazione remota

```
public String sayHello(String name) {

   String greeting = null;
   try {
      greeting = restTemplate.getForObject(
                     helloServiceUri + "/hello/{name}",
                     String.class,
                     name);     // bloccante
   } catch (RestClientException e) {
      ... REST call failed ...
   }
   return greeting;

}
```


- Client Python

- Come ulteriore alternativa, ecco un semplice client Python – basato sulla libreria **httpx**

```
#!/usr/bin/python3

import httpx

base_uri = 'http://localhost:8080/hello/'
name = 'Luca'

greeting = httpx.get(base_uri+name).text

print(greeting)
```

- Discussione

- Ecco alcune considerazioni su REST
  - consente di esporre un servizio e di invocarlo remotamente
  - non è necessario specificare in modo formale l'interfaccia del servizio
    - tuttavia, quest'interfaccia va in qualche modo definita
  - nell'architettura esagonale, è necessario realizzare gli adapter inbound e outbound per il servizio
    - in questo caso, abbiamo utilizzato il supporto e le librerie di Spring – per il controller REST (lato server) e per il client HTTP (lato client)
  - un'invocazione remota può terminare con un'eccezione (nel caso di WebClient, con un'eccezione WebClientException)
    - queste eccezioni restituiscono in genere lo stato delle risposte HTTP – ad es., *HTTP 404 Not Found* oppure *HTTP 500 Internal Server Error*

Discussione

- La semantica di REST è quella di HTTP – che usa TCP come trasporto di protocollo "affidabile"
  - pertanto, "each HTTP request message sent by a client process *eventually* arrives intact at the server – similarly, each HTTP response message sent by the server process *eventually* arrives intact at the client" [Kurose-Ross]
    - la semantica di HTTP/REST è dunque *at-most once* (*maybe*)
  - inoltre bisogna fare attenzione alla presenza di eventuali intermediari tra client e server (proxy, cache, circuit breaker o affini)
    - infatti, questi intermediari potrebbero interferire con la semantica dell'invocazione remota

Discussione

- Ulteriori considerazioni su REST
  - convenzionalmente, le operazioni di tipo GET non dovrebbero provocare cambiamenti di stato sul server – invece le operazioni di tipo POST e PATCH in genere lo cambiano, e così pure quelle di tipo PUT e DELETE (che dovrebbero essere idempotenti)
  - il server esegue le operazioni remote in modo concorrente
    - il server REST vive in un processo distinto da quello dei client REST – che possono essere molti, e accedere al server in modo concorrente
    - ciascuna diversa invocazione remota viene eseguita in modo concorrente nell'ambito di un thread (lato server) differente e separato – attenzione dunque a possibili interferenze nell'esecuzione di operazioni concorrenti

Discussione

- Con REST, client e server possono comunicare utilizzando direttamente le strutture di dati usate nel dominio delle applicazioni client e server – tuttavia, questa non è una buona pratica
  - piuttosto, è preferibile applicare anche in questo caso il pattern *Presentation Model* (discusso in una precedente dispensa) – ovvero, basare la comunicazione tra servizi distribuiti su delle "rappresentazioni" ("modello di presentazione") degli oggetti di dominio specifiche per la comunicazione – e differenti dagli oggetti di dominio stessi ("modello di dominio")
  - ciascun adapter deve poi effettuare una trasformazione tra il modello di dominio del proprio servizio e il modello di presentazione usato nella comunicazione con un altro servizio
  - utilizzeremo un presentation model nel prossimo esempio

- Il servizio restaurant-service

- Consideriamo ora il servizio **restaurant-service** per la gestione di un insieme di ristoranti – nell'ambito di un'applicazione **efood** per la gestione di un servizio di ordinazione e spedizione a domicilio di pasti da ristoranti, su scala nazionale – già introdotto in una dispensa precedente
  - la gestione dei ristoranti avviene tramite il servizio **RestaurantService**
  - i ristoranti sono definiti come un'entità JPA **Restaurant** – con attributi **id**, **name** e **location**
  - internamente al servizio, i ristoranti vengono acceduti da una base di dati mediante un repository **RestaurantRepository**

Il servizio RestaurantService

- La gestione dei ristoranti avviene tramite il servizio **RestaurantService**, con le seguenti operazioni

```
package asw.efood.restaurantservice.domain;

import ...

@Service
public class RestaurantService {

   ...

   public Restaurant createRestaurant(String name, String location) {
      ...
   }

   public Restaurant getRestaurant(Long id) { ... }

   public Collection<Restaurant> getAllRestaurants() { ... }

}
```

- Definizione dell'interfaccia del servizio

- L'interfaccia del servizio può essere definita come segue
  - operazione getRestaurant
    - GET /rest/restaurants/{restaurantId}
    - il parametro restaurantId è un intero, nel path
    - il risultato è un oggetto con la seguente struttura

```
{
   "restaurantId" : 42,
   "name" : "Baffetto",
   "location" : "Roma"
}
```


Definizione dell'interfaccia del servizio

- L'interfaccia del servizio può essere definita come segue
  - operazione getAllRestaurants
    - GET /rest/restaurants
    - nessun parametro
    - il risultato è un oggetto con la seguente struttura

```
{
   "restaurants" : [
      {
         "restaurantId" : 42,
         "name" : "Baffetto",
         "location" : "Roma"
      },
      {
         "restaurantId" : 48,
         "name" : "L'Omo",
         "location" : "Roma"
      }
   ]
}
```

Definizione dell'interfaccia del servizio

- L'interfaccia del servizio può essere definita come segue
  - operazione createRestaurant
    - POST /rest/restaurants
    - l'operazione ha come parametro un oggetto, nel corpo, con la seguente struttura

```
{
   "name" : "Seta",
   "location" : "Milano"
}
```

    - il risultato è un oggetto con la seguente struttura

```
{
   "restaurantId" : 98
}
```


- Presentation model

- Definiamo un package asw.efood.restaurantservice.api.rest per le strutture di dati richieste dall'interfaccia del servizio – può essere condiviso tra il server e i suoi client Java – con le seguenti classi
  - in corrispondenza alla risposta dell'operazione getRestaurant

```
package asw.efood.restaurantservice.api.rest;

public class GetRestaurantResponse {

   private Long restaurantId;
   private String name;
   private String location;

    … costruttori e metodi get, set e toString …

}
```

Presentation model

- Definiamo un package asw.efood.restaurantservice.api.rest per le strutture di dati richieste dall'interfaccia del servizio – può essere condiviso tra il server e i suoi client Java – con le seguenti classi
  - in corrispondenza alla risposta dell'operazione getAllRestaurants

```
public class GetRestaurantsResponse {

   private Collection<GetRestaurantResponse> restaurants;

    … costruttori e metodi get, set e toString …

}
```


Presentation model

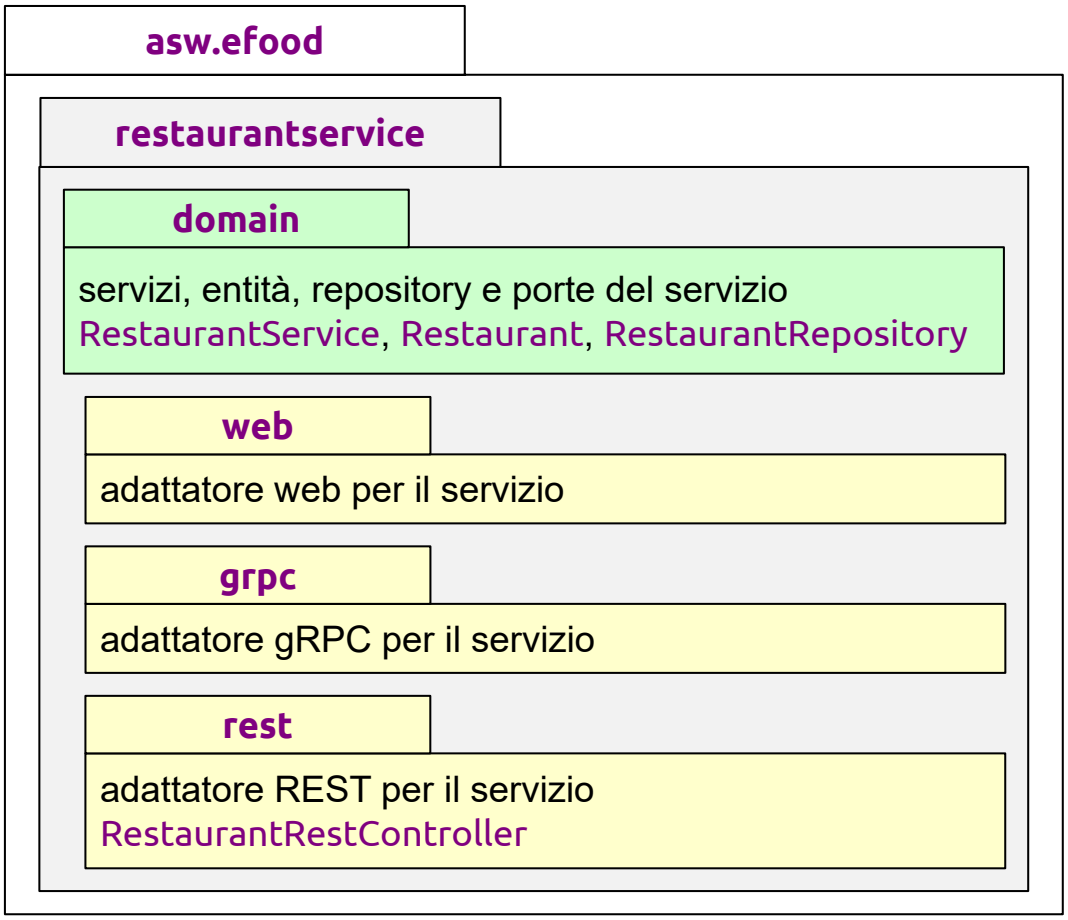
- Definiamo un package asw.efood.restaurantservice.api.rest per le strutture di dati richieste dall'interfaccia del servizio – può essere condiviso tra il server e i suoi client Java – con le seguenti classi
  - in corrispondenza alla richiesta e alla risposta dell'operazione createRestaurant

```
public class CreateRestaurantRequest {

   private String name;
   private String location;

    … costruttori e metodi get, set e toString …

}

public class CreateRestaurantResponse {

    private Long restaurantId;

    … costruttori e metodi get, set e toString …

}
```

Presentation model

- Definiamo un package asw.efood.restaurantservice.api.rest per le strutture di dati richieste dall'interfaccia del servizio – può essere condiviso tra il server e i suoi client Java – con le seguenti classi
  - è utile osservare che il framework Spring consente di effettuare automaticamente la conversione tra queste classi e le strutture di dati JSON mostrate in precedenza, in entrambi i versi – vengono utilizzate le librerie Jackson 2 per JSON
  - negli adattatori, dovremo invece occuparci di effettuare la conversione tra domain model e presentation model

- Architettura esagonale del servizio



- Server REST

- Lato server, la classe RestaurantRestController è il controller REST che implementa l'adattatore inbound per REST per il nostro servizio

```
package asw.efood.restaurantservice.rest;

import asw.efood.restaurantservice.domain.*;
import asw.efood.restaurantservice.api.rest.*;

import org.springframework.web.bind.annotation.*;

import ...

@RestController
@RequestMapping(path="/rest")
public class RestaurantRestController {

    ... vedi dopo ...

}
```

  - l'annotazione **@RequestMapping**, applicata al controller, specifica il path di base (da usare come prefisso) per tutte le operazioni

Server REST

- Nella classe RestaurantRestController è necessaria l'iniezione del servizio RestaurantService

```
@Autowired
private RestaurantService restaurantService;
```

  - nell'implementazione delle operazioni del servizio, qui nel seguito, si noti l'adattamento tipico di un adapter (in questo caso lato server) – nonché l'adattamento tra il modello di dominio interno e il modello di presentazione usato con REST (in parte svolto in modo trasparente da Spring)

Server REST

- L'operazione getRestaurant

```
@GetMapping("/restaurants/{restaurantId}")
public GetRestaurantResponse getRestaurant(
                    @PathVariable Long restaurantId) {

   Restaurant restaurant =
         restaurantService.getRestaurant(restaurantId);
   GetRestaurantResponse response =
         restaurantToGetRestaurantResponse(restaurant);
   return response;

}

private GetRestaurantResponse restaurantToGetRestaurantResponse(
                                    Restaurant restaurant) {

   return new GetRestaurantResponse(
                    restaurant.getId(),
                    restaurant.getName(),
                    restaurant.getLocation());

}
```


Server REST

- L'operazione getRestaurant (variante)
  - restituisce il codice di stato HTTP 404 (anziché 500) se il ristorante non viene trovato

```
@GetMapping("/restaurants/{restaurantId}")
public GetRestaurantResponse getRestaurant(
                    @PathVariable Long restaurantId) {

   Restaurant restaurant =
         restaurantService.getRestaurant(restaurantId);
   if (restaurant!=null) {
      GetRestaurantResponse response =
            restaurantToGetRestaurantResponse(restaurant);
      return response;
   } else {
      throw new ResponseStatusException(
         HttpStatus.NOT_FOUND, "Restaurant not found"
      );
   }

}
```

Server REST

- L'operazione getAllRestaurants

```
@GetMapping("/restaurants")
public GetRestaurantsResponse getRestaurants() {

   Collection<Restaurant> restaurants =
                     restaurantService.getAllRestaurants();
   Collection<GetRestaurantResponse> restaurantResponses =
         restaurants
             .stream()
             .map(r -> restaurantToGetRestaurantResponse(r))
             .collect(Collectors.toList());
   return new GetRestaurantsResponse(restaurantResponses);

}
```


Server REST

- L'operazione createRestaurant

```
@PostMapping("/restaurants")
public CreateRestaurantResponse createRestaurant(
             @RequestBody CreateRestaurantRequest request) {

   String name = request.getName();
   String location = request.getLocation();
   Restaurant restaurant =
         restaurantService.createRestaurant(name, location);
   CreateRestaurantResponse response =
         new CreateRestaurantResponse(restaurant.getId());
   return response;

}
```

- Client REST

- Il lato client è un'altra applicazione o servizio che vuole accedere, in questo esempio, al servizio **restaurant-service**
  - supponiamo che sia un'altra applicazione Spring Boot, il cui package di base è asw.efood.samplerestaurantclient
  - nel package asw.efood. samplerestaurantclient.domain va definita un'interfaccia richiesta (una porta) RestaurantClientPort per accedere al servizio dei ristoranti
  - poi, applicando l'architettura esagonale, va definito un outbound adapter per accedere al servizio dei ristoranti
    - a tal fine, definiamo il package asw.efood.samplerestaurantclient.restaurantclient.rest con la classe RestaurantClientRestAdapter

Architettura esagonale del servizio

- Architettura esagonale del servizio **sample-restaurant-client**

Client REST

- Lato client, l'interfaccia RestaurantClientPort

```
package asw.efood.samplerestaurantclient.domain;

import java.util.*;

public interface RestaurantClientPort {

    Long createRestaurant(String name, String location);
    Restaurant getRestaurant(Long restaurantId);
    List<Restaurant> getAllRestaurants();

}
```

  - nello stesso package, definiamo anche una classe Restaurant – che è diversa dall'entità Restaurant nel dominio del server

```
public class Restaurant {

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

    … costruttori e metodi get, set e toString …

}
```


Client REST

- Anche in questo caso, che cosa faccia il client finale di questo servizio è poco rilevante per la nostra discussione
  - ecco una porzione di esempio del client – si noti la dipendenza dall'interfaccia/porta RestaurantClientPort – e soprattutto che il client non dipende in alcun modo dall'adozione di REST

```
package asw.efood.samplerestaurantclient.domain;

import ...;

@Component
public class RestaurantClientRunner implements CommandLineRunner {

   @Autowired
   private RestaurantClientPort restaurantClientAdapter;

   public void run(String[] args) {

      ... restaurantClientAdapter.getRestaurant(42L) ...

   }

}
```

Client REST

- Lato client, la classe RestaurantClientRestAdapter

```
package asw.efood.samplerestaurantclient.restaurantclient.rest;

import asw.efood.samplerestaurantclient.domain.*;

import asw.efood.restaurantservice.api.rest.*;

import ...;

@Service
public class RestaurantClientRestAdapter
                implements RestaurantClientPort {

   ... vedi dopo ...

}
```

  - si noti, nell'implementazione della classe, l'adattamento tipico di un adapter (in questo caso outbound, lato client) – nonché l'adattamento tra il modello di dominio interno e il modello di presentazione usato con REST (in parte svolto in modo trasparente da Spring)

Client REST

- Alcune definizioni preliminari nella classe RestaurantClientRestAdapter

```
@Value("${asw.efood.restaurantservice.rest.uri}")
private String restaurantServiceUri;
```

```
# application.properties
asw.efood.restaurantservice.rest.uri=http://localhost:8080/rest
```

```
private WebClient webClient;

public RestaurantClientRestAdapter(
                              WebClient.Builder webClientBuilder) {
   this.webClient =
         webClientBuilder.baseUrl(restaurantServiceUri).build();
}
```

Client REST

- Il metodo getRestaurant

```
public Restaurant getRestaurant(Long restaurantId) {

   Restaurant restaurant = null;
   String restaurantUri = restaurantServiceUri +
                          "/restaurants/{restaurantId}";
   Mono<GetRestaurantResponse> response = webClient
                    .get()
                    .uri(restaurantUri, restaurantId)
                    .retrieve()
                    .bodyToMono(GetRestaurantResponse.class);
   try {
      GetRestaurantResponse grr = response.block();
      if (grr!=null) {
         restaurant = getRestaurantResponseToRestaurant(grr);
      }
   } catch (WebClientException e) { ... REST call failed ... }
   return restaurant;

}
```


Client REST

- Il metodo di supporto getRestaurantResponseToRestaurant

```
private Restaurant getRestaurantResponseToRestaurant(
                                      GetRestaurantResponse r) {

   return new Restaurant(
                    r.getRestaurantId(),
                    r.getName(),
                    r.getLocation());

}
```

Client REST

- Il metodo getAllRestaurants

```
public List<Restaurant> getAllRestaurants() {

   List<Restaurant> restaurants = null;
   String restaurantsUri = restaurantServiceUri + "/restaurants";
   Mono<GetRestaurantsResponse> response = webClient
                    .get()
                    .uri(restaurantsUri)
                    .retrieve()
                    .bodyToMono(GetRestaurantsResponse.class);
   try {
      GetRestaurantsResponse grr = response.block();
      if (grr!=null) {
         restaurants = grr.getRestaurants().stream()
             .map( r -> getRestaurantResponseToRestaurant(r) )
             .collect(Collectors.toList());
      }
   } catch (WebClientException e) { ... REST call failed ... }
   return restaurants;

}
```


Client REST

- Il metodo createRestaurant

```
public Long createRestaurant(String name, String location) {

   Long restaurantId = null;
   String restaurantUri = restaurantServiceUri + "/restaurants";
   CreateRestaurantRequest request =
                    new CreateRestaurantRequest(name, location);
   Mono<CreateRestaurantResponse> response = webClient
                    .post()
                    .uri(restaurantUri)
                    .body(BodyInserters.fromValue(request))
                    .retrieve()
                    .bodyToMono(CreateRestaurantResponse.class);
   try {
      CreateRestaurantResponse crr = response.block();
      if (crr!=null) {
         restaurantId = crr.getRestaurantId() ;
      }
   } catch (WebClientException e) { ... REST call failed ... }
   return restaurantId;

}
```

- Client Python

- Come alternativa, viene ora mostrato un client Python semplificato
  - basato sulle librerie **httpx** (per il client HTTPS) e **typing** e **pydantic** (per le classi per le strutture di dati)

```
import httpx

from typing import List
from pydantic import BaseModel
```

  - ecco la classe di dominio per i ristoranti

```
class Restaurant(BaseModel):

   rid: int
   name: str
   location: str

   def __str__(self):
      return "Restaurant(rid=" + str(self.rid) +   \
         ", name=" + self.name + ", location=" + self.location + ")"
```


Client Python

- Come alternativa, viene ora mostrato un client Python semplificato
  - ecco le classi del presentation model (semplificato)

```
class GetRestaurantResponse(BaseModel):
   restaurantId: int
   name: str
   location: str

class GetRestaurantsResponse(BaseModel):
   restaurants: List[GetRestaurantResponse]
```

Client Python

- Come alternativa, viene ora mostrato un client Python semplificato
  - ecco l'adattatore per il client REST (semplificato)

```
class RestaurantClient:

   def get_restaurant(restaurant_id: int):
      request_url =  \
         'http://localhost:8080/rest/restaurants/{id}'   \
            .format(id=restaurant_id)
      response = httpx.get(request_url)
      restaurant_response =  \
         GetRestaurantResponse.parse_obj(response.json())
      restaurant =   \
         Restaurant(rid=restaurant_response.restaurantId,
            name=restaurant_response.name,
            location=restaurant_response.location)
      return restaurant
```


Client Python

- Come alternativa, viene ora mostrato un client Python semplificato
  - ecco l'adattatore per il client REST (semplificato)

```
   def get_all_restaurants():
      request_url = 'http://localhost:8080/rest/restaurants'
      response = httpx.get(request_url)
      restaurants_response =  \
         GetRestaurantsResponse.parse_obj(response.json())
      restaurants: list[Restaurant] = []
      for restaurant_response in restaurants_response.restaurants:
         restaurants.append(
            Restaurant(rid=restaurant_response.restaurantId,
               name=restaurant_response.name,
               location=restaurant_response.location) )
      return restaurants
```

Client Python

- Come alternativa, viene ora mostrato un client Python semplificato
  - ecco un client REST (semplificato)

```
from restaurant_python_client.restaurant_rest_python_client \
   import RestaurantClient

# trova il primo ristorante
r = RestaurantClient.get_restaurant(1)
print(r)

# trova tutti i ristoranti
rr = RestaurantClient.get_all_restaurants()
print(rr)

# trova l'ultimo ristorante
last_id = rr[-1].rid
last_r = RestaurantClient.get_restaurant(last_id)
print(last_r)
```


- Esercizio

- In un precedente esercizio è stato richiesto di estendere il servizio per la gestione dei ristoranti con la gestione dei menu dei ristoranti
  - nel dominio del servizio dei ristoranti, il menu **RestaurantMenu** di un ristorante **Restaurant** è un elenco di **MenuItem** (ciascuno con un id, un nome e un prezzo)
  - andavano definite un'operazione per trovare il menu di un ristorante e un'operazione per creare il menu di un ristorante
  - qui si chiede
    - lato server, di esporre queste funzionalità mediante REST
    - lato client, di invocare queste funzionalità
    - si supponga che, nel dominio del client, un ristorante **Restaurant** abbia un elenco di **MenuItem** (ciascuno con un id, una descrizione e un prezzo) – il domain model del server e del client possono essere strutturati in modo differente

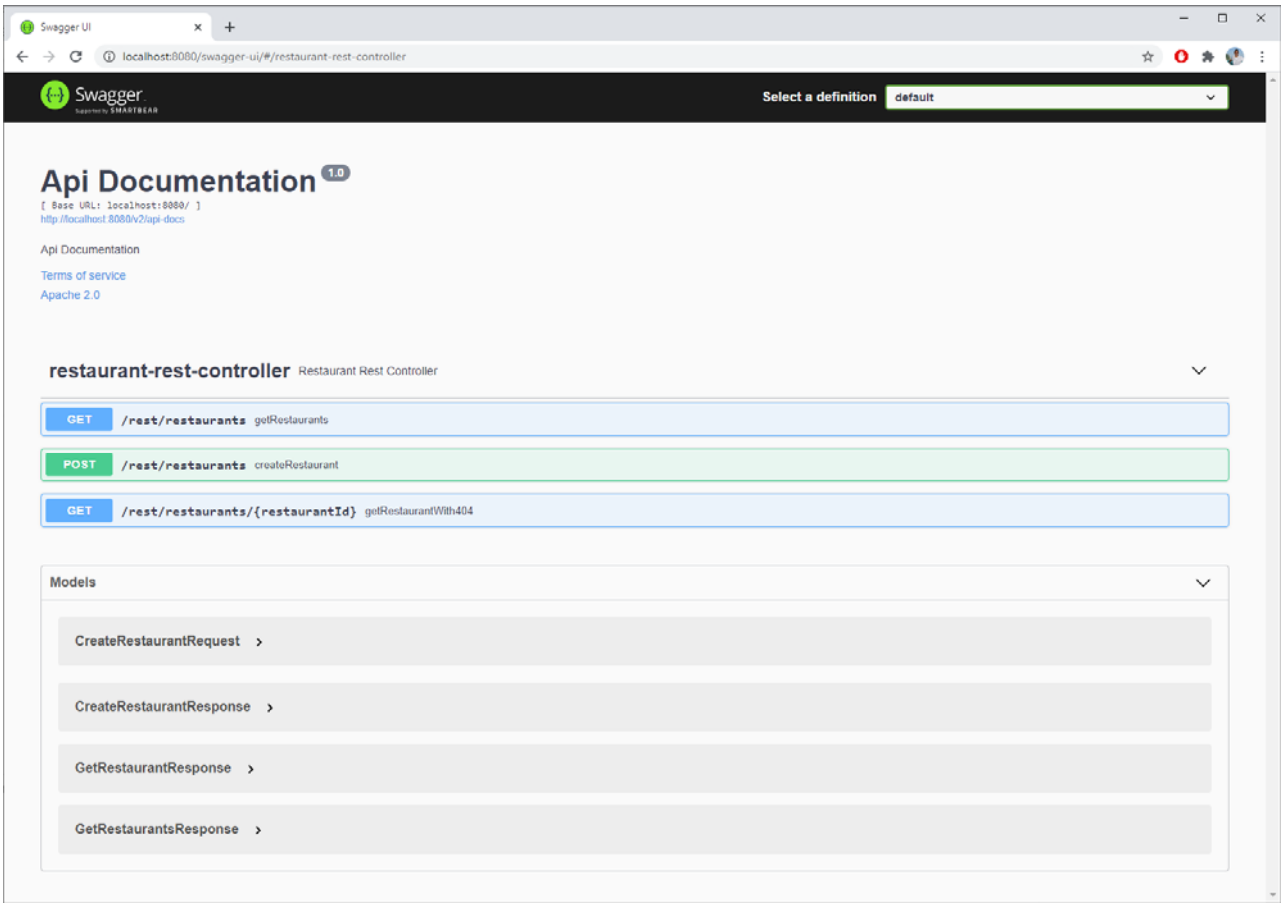
- OpenAPI Specification e Swagger UI

- La *OpenAPI Specification* (https://www.openapis.org/) definisce una descrizione standard delle API REST
  - questa descrizione consente sia alle persone che ai client software di scoprire e comprendere facilmente l'interfaccia di un servizio – senza la necessità di guardare il codice del servizio
- Inoltre, *Swagger UI* (https://swagger.io/) consente di visualizzare e interagire con un servizio REST mediante una semplice interfaccia web che viene generata automaticamente a partire dall'interfaccia del servizio REST

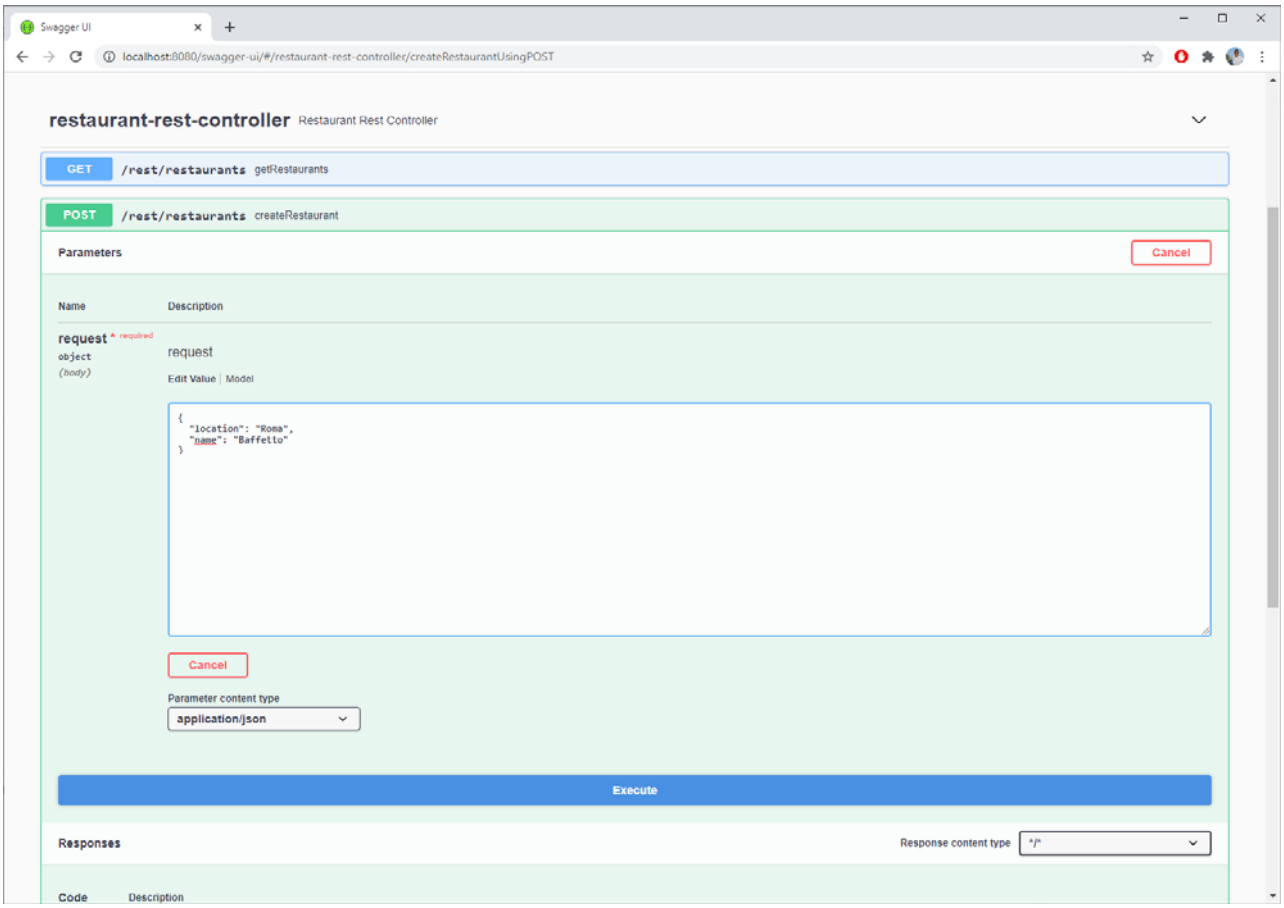
OpenAPI Specification e Swagger UI

- Con Spring Boot, l'integrazione con OpenAPI e Swagger è basata sul progetto SpringDoc OpenAPI
  - nell'applicazione server, è sufficiente utilizzare la dipendenza starter org.springdoc:springdoc-openapi-starter-webmvc-ui
  - questo genera automaticamente un adapter web per il servizio – che è raggiungibile alla pagina /swagger-ui/index.html del server, ad es., all'indirizzo http://localhost:8080/swagger-ui/
  - questa interfaccia web è utile soprattutto durante lo sviluppo, per eseguire dei semplici test manuali

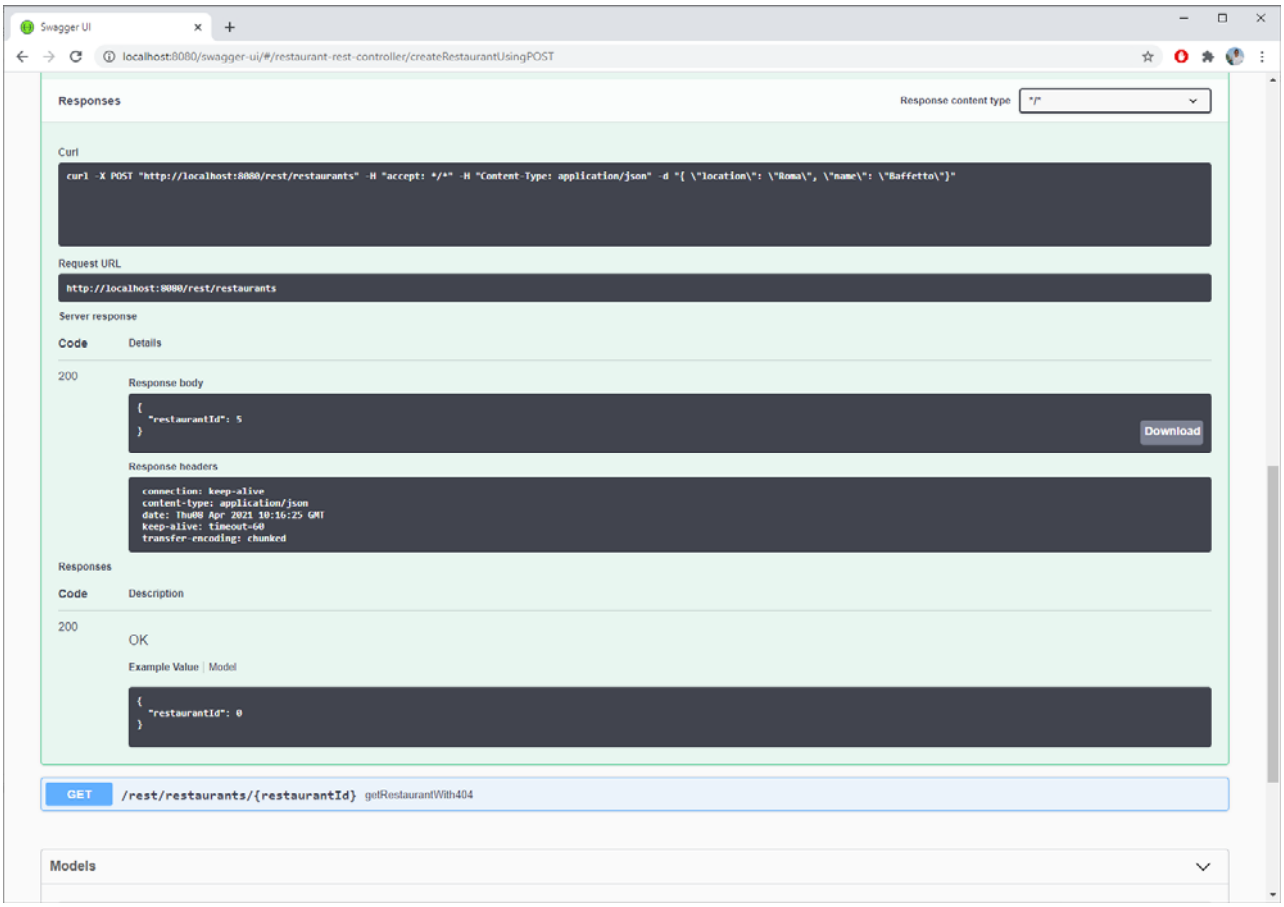
Swagger UI



Swagger UI



Swagger UI



* Invocazioni remote asincrone

- Negli esempi mostrati, i client invocano le operazioni remote in modo sincrono
  - tuttavia, è anche possibile effettuare le invocazioni remote in modo asincrono
    - questo è utile soprattutto per operazioni di lunga durata, e per consentire di svolgere più attività in modo concorrente, e per ovviare alla latenza introdotta dall'invocazione remota
    - non sono le operazioni del server ad essere asincrone – è il client che invoca le operazioni del server in modo asincrono
  - esistono diverse possibilità
    - ne esaminiamo qui una abbastanza generale, basata sulla libreria java.util.concurrent di Java
    - un'altra soluzione, più specifica, consiste nell'usare direttamente i tipi **Mono** e **Flux** di Reactor (https://projectreactor.io/)

@Async e CompletableFuture<V>

- Descriviamo ora un modello di programmazione generale per operazioni asincrone
  - è basato sulla libreria java.util.concurrent di Java, e in particolare sull'annotazione **@Async** e sulla classe **CompletableFuture<V>**
  - si tratta di un modello generale, utile nella programmazione concorrente – può essere utilizzato anche per le invocazioni remote asincrone, ma non è limitato solo alle invocazioni remote
  - è importante capire che, nell'utilizzo che proponiamo di questo modello per l'invocazione remota asincrona
    - non sono le operazioni del server ad essere asincrone
    - piuttosto, è il client che invoca le operazioni del server in modo asincrono anziché sincrono

@Async e CompletableFuture<V>

- Ecco un modello di programmazione per operazioni asincrone
  - l'annotazione **@Async** indica un'operazione asincrona
    - può essere utilizzata in un'operazione del client che racchiude un'invocazione remota, per effettuare quell'invocazione remota in modo asincrono
    - intuitivamente, l'operazione asincrona viene eseguita in un thread separato rispetto a quello del suo chiamante
    - per abilitare le operazioni asincrone, la classe per l'applicazione Spring Boot deve essere annotata con **@EnableAsync**
    - alcune osservazioni su **@Async**
      - va usata solo con metodi pubblici
      - non funziona se l'operazione asincrona viene invocata da un altro metodo definito nella stessa classe

@Async e CompletableFuture<V>

- Ecco un modello di programmazione per operazioni asincrone
  - la classe **CompletableFuture<V>** può essere usata come tipo di ritorno di un'operazione asincrona che deve restituire un valore di tipo **V**
    - l'operazione da eseguire in modo asincrono *op()* può essere invocata come CompletableFuture.completedFuture(*op()*)
    - al momento dell'invocazione di un'operazione asincrona, al client viene immediatamente restituito un oggetto di tipo **CompletableFuture<V>** – che poi offre queste operazioni
      - **V get()** (bloccante) – per ottenere il valore restituito
      - **boolean isDone()** – per sapere se l'esecuzione dell'operazione asincrona è terminata normalmente
      - **CompletableFuture.allOf(*cf1*, *cf2*, …).join()** – attende la terminazione delle invocazioni ***cf1***, ***cf2***, …
      - **cancel(true)** – per chiedere la cancellazione dell'operazione asincrona

Cambiamenti al client REST

- Introduciamo una nuova interfaccia asincrona per la porta per l'accesso al servizio dei ristoranti

```
package asw.efood.samplerestaurantclient.domain;

import java.util.*;
import java.util.concurrent.CompletableFuture;

public interface RestaurantClientAsyncPort {

   CompletableFuture<Long>
               createRestaurantAsync(String name, String location);
   CompletableFuture<Restaurant> getRestaurantAsync(Long restaurantId);
   CompletableFuture<List<Restaurant>> getAllRestaurantsAsync();

}
```

Cambiamenti al client REST

- L'adattatore REST lato client per questa interfaccia asincrona
  - per semplicità, le invocazioni remote vengono effettuate mediante l'adattatore REST sincrono visto in precedenza

```
package asw.efood.samplerestaurantclient.restaurantclient.rest;

import asw.efood.samplerestaurantclient.domain.*;

import ...;

import java.util.concurrent.CompletableFuture;
import org.springframework.scheduling.annotation.Async;

@Service
public class RestaurantClientAsyncRestAdapter
                implements RestaurantClientAsyncPort {

   @Autowired
   private RestaurantClientPort restaurantClientAdapter;

   ... vedi dopo ...

}
```


Cambiamenti al client REST

- L'adattatore REST per questa interfaccia asincrona
  - il metodo getRestaurantAsync – incapsula la chiamata all'operazione remota sincrona getRestaurant

```
@Async
public CompletableFuture<Restaurant>
                    getRestaurantAsync(Long restaurantId) {

   return CompletableFuture.completedFuture(
         restaurantClientAdapter.getRestaurant(restaurantId)
      );

}
```

  - gli altri metodi sono simili

Cambiamenti al client REST

- In questo caso bisogna però cambiare anche il client finale del servizio – perché usa invocazioni asincrone
  - ecco una porzione di esempio del client – si noti che il client non dipende in alcun modo dall'adozione di REST

```
@Component
public class RestaurantAsyncClientRunner implements CommandLineRunner {

   @Autowired
   private RestaurantClientAsyncPort restaurantClientAsyncAdapter;

   public void run(String[] args) {

      CompletableFuture<Restaurant> future =
            restaurantClientAsyncAdapter.getRestaurantAsync(42L);
      ...
      Restaurant restaurant = future.get();

   }

}
```


Discussione

- Le operazioni asincrone e le invocazioni asincrone sono utili in caso di operazioni di lunga durata – oppure per implementare richieste di tipo *send-and-forget*
  - esistono diversi modi per implementare operazioni asincrone e gestire invocazioni asincrone
    - ne abbiamo mostrato uno, in cui, intuitivamente, l'invocazione di un'operazione asincrona avviene in un thread separato
  - le operazioni asincrone possono essere utilizzate, in particolare, anche per effettuare invocazioni remote asincrone
  - le operazioni asincrone abilitano anche l'esecuzione concorrente di più operazioni asincrone
    - questo è utile, ad esempio, per evitare di sommare le latenze di più invocazioni remote, se queste possono avvenire in modo concorrente anziché in modo sequenziale

* Discussione

- In questa dispensa abbiamo presentato REST come una tecnologia per l'invocazione remota, basata su HTTP
  - l'uso di REST richiede
    - la definizione dell'interfaccia del servizio da esporre remotamente – in genere in modo informale
    - la scrittura, per il servizio, di un adapter inbound lato server (un controller REST) e, per un servizio client, di un adapter outbound lato client (scritto con riferimento a un client HTTP generico)
    - è di solito opportuno codificare i dati scambiati tra server e client mediante un opportuno Presentation Model – un insieme di classi "rappresentazione"
    - i dati vengono scambiati su HTTP mediante JSON oppure XML – Spring e altri framework supportano in genere una conversione automatica (anche con la possibilità di qualche personalizzazione della conversione)